

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

DTIC FILE COPY

0

AD-A196 541

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/CI/NR 88- 58	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A CLASSIFICATION METHODOLOGY AND RETRIEVAL MODEL TO SUPPORT SOFTWARE REUSE		5. TYPE OF REPORT & PERIOD COVERED <input checked="" type="checkbox"/> THESIS
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) DANIEL LEE RUBLE		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: TEXAS A & M UNIVERSITY		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE 1988
		13. NUMBER OF PAGES 159
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) AFIT/NR Wright-Patterson AFB OH 45433-6583		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) DISTRIBUTED UNLIMITED: APPROVED FOR PUBLIC RELEASE		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) SAME AS REPORT		
18. SUPPLEMENTARY NOTES Approved for Public Release: IAW AFR 190-1 LYNN E. WOLAVER <i>Lynn Wolaver</i> Dean for Research and Professional Development Air Force Institute of Technology Wright-Patterson AFB OH 45433-6583 19 July 88		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) ATTACHED		

DTIC

S AUG 04 1988 D  
HDD FORM 1473  
1 JAN 73

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

## ABSTRACT

### A Classification Methodology and Retrieval Model to Support Software Reuse (December 1987)

Daniel Lee Ruble, A.A., Del Mar College;

B.S., Texas A&I University;

M.C.S., Texas A&M University

Co-Chairs of Advisory Committee: Dr. Sallie V. Sheppard  
Dr. William M. Lively

Studies have shown that reusing existing software can reduce development costs, speed up the development process, and provide a more reliable product. A software classification methodology and retrieval model have been developed to support the organization and location of reusable software components. This capability required the design and development of three cooperating processes: (1) an organization for the reusable software collection, (2) a method for describing software components, and (3) a mechanism to access (locate and retrieve) the desired software component.

A faceted classification scheme and retrieval model were designed to overcome the deficiencies found in existing software reuse systems. These deficiencies include a lack of retrieval support for the developer, static classification schemes based on enumerative techniques which are difficult to expand or modify, and limited descriptive capabilities based on keyword retrieval technology. The retrieval model resulting from this research is one of the key components needed for large-scale software reuse.

The faceted classification model from library science was used to design a software classification methodology based on an analysis and synthesis process. An analysis of the reusable software components is used to construct the classification. A synthesis process can then be used to describe items in the collection. The faceted methodology is adaptable to changes and growth in the target collection, concise in its descriptive format, facilitates automation, and supports citation order changes to adapt the organization to different users. Application of the faceted classification methodology to a test collection is presented.

A formal retrieval model was designed using a combination of techniques from the boolean and vector space information retrieval models. The retrieval model uses attribute tuples to represent both the software components and user queries. The retrieval mechanism combines direct attribute matching from the boolean model with a similarity heuristic to provide relevance estimation. The general principles used to design the classification methodology and retrieval model were informally verified by experience with a rapid prototype system constructed as part of the research. Based on these initial experiences, the faceted classification methodology and hybrid retrieval mechanism appear to provide an effective retrieval system for reusable software components.



Accession For	
NTIS DATA	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
Distribution/	
Availability Codes	
Dist	
A-1	

**A CLASSIFICATION METHODOLOGY AND RETRIEVAL  
MODEL TO SUPPORT SOFTWARE REUSE**

A Dissertation  
by  
DANIEL LEE RUBLE

Submitted to the Graduate College of  
Texas A&M University  
in partial fulfillment of the requirements for the degree  
DOCTOR OF PHILOSOPHY

December 1987

Major Subject: Computer Science

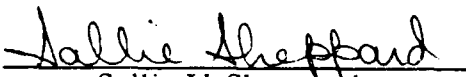
**A CLASSIFICATION METHODOLOGY AND RETRIEVAL  
MODEL TO SUPPORT SOFTWARE REUSE**

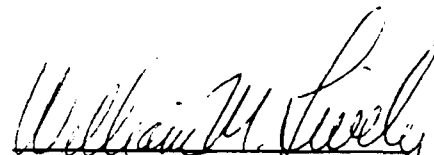
A Dissertation

by

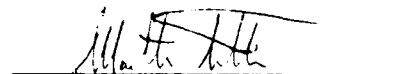
**DANIEL LEE RUBLE**

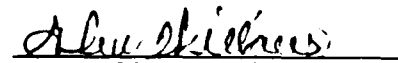
Approved as to style and content by:

  
Sallie V. Sheppard  
(Co-Chair of Committee)

  
William M. Lively  
(Co-Chair of Committee)

  
W. Homer Carlisle  
(Member)

  
Marietta J. Tretter  
(Member)

  
Glen Williams  
(Head of Department)

December 1987

## ABSTRACT

### A Classification Methodology and Retrieval Model to Support Software Reuse (December 1987)

Daniel Lee Ruble, A.A., Del Mar College;

B.S., Texas A&I University;

M.C.S., Texas A&M University

Co-Chairs of Advisory Committee: Dr. Sallie V. Sheppard  
Dr. William M. Lively

Studies have shown that reusing existing software can reduce development costs, speed up the development process, and provide a more reliable product. A software classification methodology and retrieval model have been developed to support the organization and location of reusable software components. This capability required the design and development of three cooperating processes: (1) an organization for the reusable software collection, (2) a method for describing software components, and (3) a mechanism to access (locate and retrieve) the desired software component.

A faceted classification scheme and retrieval model were designed to overcome the deficiencies found in existing software reuse systems. These deficiencies include a lack of retrieval support for the developer, static classification schemes based on enumerative techniques which are difficult to expand or modify, and limited descriptive capabilities based on keyword retrieval technology. The retrieval model resulting from this research is one of the key components needed for large-scale software reuse.

→ The faceted classification model from library science was used to design a software classification methodology based on an analysis and synthesis process. An analysis of the reusable software components is used to construct the classification. A synthesis process can then be used to describe items in the collection. The faceted methodology is adaptable to changes and growth in the target collection, concise in its descriptive format, facilitates automation, and supports citation order changes to adapt the organization to different users. Application of the faceted classification methodology to a test collection is presented.

A formal retrieval model was designed using a combination of techniques from the boolean and vector space information retrieval models. The retrieval model uses attribute tuples to represent both the software components and user queries. The retrieval mechanism combines direct attribute matching from the boolean model with a similarity heuristic to provide relevance estimation. The general principles used to design the classification methodology and retrieval model were informally verified by experience with a rapid prototype system constructed as part of the research. Based on these initial experiences, the faceted classification methodology and hybrid retrieval mechanism appear to provide an effective retrieval system for reusable software components.

→ keywords: computer programs, information retrieval, productivity, cost effectiveness, output, time, user, manual, documentation, software, etc.

(25)

To Pat, Robin, and Christy



## ACKNOWLEDGEMENTS

First, and foremost, I would like to thank Dr. Sallie Sheppard and Dr. William Lively for their support, guidance and encouragement throughout my graduate program at Texas A&M. They were always there to listen and advise, during the best and worst of times. I wish to thank Dr. Homer Carlisle and Dr. Marietta Tretter for serving on my committee and always being available for counsel. A special thanks goes to Dr. John McInroy for his support as an adjunct committee member and to Dr. James Smith for representing the Graduate Council.

I would like to thank all my friends and colleagues here at Texas A&M for their support, helpful suggestions, and warm friendship, especially Dave Umphress, Scott Teel, Marcus Brown, Craig Murra, Steve Moffett, and Al Crawford.

Finally, I would like to thank my best friend and wife, Pat, and my daughters, Christy and Robin, for their love and understanding. This work is dedicated to them.

## TABLE OF CONTENTS

	Page
1. INTRODUCTION.....	1
1.1 Background.....	1
1.1.1 Impetus.....	2
1.1.2 Types of Reuse.....	4
1.1.3 Current Status.....	5
1.2 Research Overview.....	7
1.3 Overview of Following Sections.....	9
2. REVIEW OF PREVIOUS WORK.....	10
2.1 Software Reusability.....	10
2.1.1 Reusable Code.....	13
2.1.1.1 Functional Collections.....	14
2.1.1.2 Libraries in Industry.....	17
2.1.1.3 Software Factories.....	20
2.1.1.4 Programming Languages and Environments.....	21
2.1.2 Application Generators.....	24
2.1.2.1 The Draco System.....	25
2.1.2.2 KBMC.....	27
2.1.3 Transformation Systems.....	27
2.1.3.1 PSI Program Synthesis System.....	31
2.1.3.2 TAMPR.....	33
2.1.3.3 Programmer's Apprentice.....	34
2.2 Information Retrieval.....	35
2.2.1 Boolean Retrieval Systems.....	36
2.2.1.1 Stairs.....	38
2.2.1.2 Liars.....	39
2.2.1.3 Fakyr.....	40
2.2.2 Vector Space Retrieval.....	41
2.2.2.1 Smart.....	42
2.2.2.2 Sire.....	42
2.2.2.3 Caliban.....	43
2.2.3 Probabilistic Information Retrieval.....	44
2.2.3.1 Harter's Model.....	45
2.2.3.2 University of Massachusetts System.....	45
2.3 Classification Schemes and Techniques.....	46
2.3.1 General Classification Terms.....	46
2.3.2 Types of Classification Systems.....	50
2.3.3 Library Classification.....	52
2.3.3.1 Dewey Decimal Classification.....	53
2.3.3.2 Universal Decimal Classification.....	55
2.3.3.3 Library of Congress.....	57
2.3.3.4 Bibliographic Classification.....	60
2.3.3.5 Colon Classification.....	62

	Page
2.3.4 Software Classification.....	64
2.3.4.1 General Schemes.....	64
2.3.4.2 Program Libraries.....	65
2.3.4.3 Software Directories.....	66
2.4 Summary and Conclusion.....	67
3. CLASSIFICATION OF SOFTWARE COMPONENTS.....	69
3.1 Design Objectives.....	70
3.1.1 General Classification Objectives.....	70
3.1.2 Software Classification Objectives.....	72
3.2 Selection of a Classification Model.....	73
3.2.1 Identification, Location, and Precision.....	74
3.2.2 Collocation, Organization, and Structure.....	76
3.2.3 Expansion and Automation.....	76
3.2.4 Selection Summary.....	78
3.3 A Faceted Scheme and Classification Methodology.....	78
3.3.1 Basic Concepts.....	79
3.3.2 The Methodology.....	79
3.3.3 The Faceted Scheme.....	82
3.3.3.1 The Analysis Phase.....	83
3.3.3.2 The Synthesis Phase.....	89
3.4 Application of the Faceted Methodology.....	90
4. SOFTWARE COMPONENT RETRIEVAL.....	93
4.1 Functional Requirements.....	93
4.1.1 Query Formulation.....	94
4.1.2 Relevance Ranking.....	95
4.1.3 Vocabulary Control.....	96
4.1.4 Collection Reordering.....	98
4.2 Design Decisions for the Retrieval Mechanism.....	98
4.2.1 The Retrieval Model.....	98
4.2.2 Query Formulation and Vocabulary Control.....	101
4.2.3 A Similarity Heuristic.....	102
4.2.4 Facet Reordering.....	104
4.3 Design of the Retrieval System.....	104
4.3.1 Control Structure.....	105
4.3.2 User Interface and Vocabulary Control.....	107
4.3.3 Phased Retrieval Mechanism.....	107
4.4 Rapid Prototype Implementation.....	110
4.4.1 Rapid Prototype Support.....	110
4.4.2 Control Structure of the Rapid Prototype.....	111
4.4.3 Support Routines.....	113

	Page
5. EXPANDING THE SOFTWARE COLLECTION.....	116
5.1 Selecting Another Software Domain.....	116
5.2 The Modification Process.....	118
5.3 Modifying the Prototype Schedule.....	119
5.4 Results and Conclusions.....	121
6. SUMMARY, CONCLUSIONS AND FUTURE WORK.....	123
6.1 Summary of the Research.....	123
6.1.1 Software Reuse.....	124
6.1.2 Faceted Methodology.....	124
6.1.3 Retrieval Mechanism.....	126
6.1.4 Rapid Prototype.....	127
6.2 Results and Conclusions.....	128
6.2.1 Rapid Prototype.....	128
6.2.2 Faceted Classification.....	130
6.2.3 Retrieval Mechanism.....	134
6.3 Future Work.....	134
6.3.1 System Enhancements.....	135
6.3.2 Additional Research.....	136
6.3.2.1 A Classification Assistant.....	136
6.3.2.2 Other Software Development Products.....	137
6.3.2.3 Reuse Environment Integration.....	137
6.3.2.4 Thesaurus Construction.....	138
6.3.2.5 Retrieval Based Upon Reuse Metrics.....	138
REFERENCES.....	140
APPENDIX 1 - SOFTWARE COMPONENT SCHEDULE 1.....	151
APPENDIX 2 - SOFTWARE COMPONENT SCHEDULE 2.....	155
VITA.....	159

## LIST OF FIGURES

Figure	Page
1. Software life cycle: (a) traditional model and (b) reuse model .....	12
2. Draco System Overview .....	26
3. A Draco Statistics Reporting Domain .....	28
4. Transformation System Overview .....	29
5. Literature Hierarchy .....	48
6. Computer Language Hierarchy .....	49
7. Faceted Classification Example .....	51
8. Enumerated Classification Example .....	51
9. Subject Scattering in DDC .....	55
10. DC18 Music Class .....	75
11. Constructing a Faceted Classification .....	80
12. Retrieval System Control Hierarchy (first level) .....	105
13. Retrieval System Control Hierarchy (second Level) .....	105
14. Retrieval System Control Logic .....	106
15. Facet-based Query Construction: (a) Form menu and (b) Activity menu .....	112
16. Ranked Output from the Prototype .....	115
17. Ranked Retrieval Output: (a) from default citation order and (b) from modified citation order .....	133

## 1. INTRODUCTION

Software is expensive. The U. S. Department of Defense alone will spend over \$30 billion per year on software by 1990 [Horowitz and Munson 1984], and Ramamoorthy et al. [1984] project the cost of software in the United States will approach 13 percent of the Gross National Product during the same time frame. These figures, while staggering, only represent a portion of the total cost of computer software. Canceled projects, faulty software, delayed software, and maintenance costs make the actual amount much higher. There are several factors contributing to this tremendous growth in software cost such as increasingly complex requirements for new software systems, a shortage of qualified software professionals, and a lack of significant improvement in software development tools and methodologies. For the foreseeable future, there is little doubt that software costs will continue to rise. Software reuse is viewed by many experts as a promising near term solution to this problem. A classification methodology and retrieval mechanism to support software reuse is the focus of the research reported herein.

### 1.1 Background

Studies have shown that a large percentage of computer applications contain common functions. A California study [Jones 1984] of commercial banking and insurance applications reported nearly 75 percent of the

---

This dissertation was prepared in the format of *ACM Computing Surveys*.

functions identified existed in more than one program. Goodell [1983] examined over 1300 application programs at the Burroughs Corporation and identified only 24 unique primary functions. The well known study of business applications at Raytheon [Lanergan and Grasso 1983, 1984] revealed that 60 percent of their programs were performing essentially similar functions.

During the last 20 years, productivity in software development has only increased by 3 to 8 percent per year. However, software engineering experts estimate that software reuse can increase productivity by at least an order of magnitude [Horowitz and Munson, 1984; Jones 1984; Lanergan and Grasso 1983, 1984; Boehm and Standish 1983; Standish 1983, 1984]. Reusing existing software systems or portions of existing software products can reduce development costs, speed up the development process, reduce testing requirements, and provide a more reliable product.

### *1.1.1 Impetus*

The concept of reusability is not new. It is a general engineering principle derived from the desire to avoid duplication and to capture commonalty in classes of similar tasks [Wegner 1984]. Wegner uses a manufacturing analogy to view software as a capital investment and explain the economic incentives for its reuse.

Machine tools of the industrial revolution and software tools such as compilers are both reusable resources. Moreover, any reusable resource may be thought of as a capital good whose development cost may be recovered over its set of uses.

Reuse can eliminate a large percentage of the development process for each reused component with a corresponding decrease in cost [Pedersen 1984; Yeh

et al. 1984a; Wegner 1984]. Thus, there are several incentives for the reuse of software components: economics, reliability, manpower, and time. Reusing software reduces the cost and manpower required by reducing the amount of specification, design, coding, and testing which must be accomplished. Maintenance, which represents the largest percentage of cost and manpower in the software life cycle, is also reduced by using components of proven quality and reliability which are well documented and easier to understand [Harrison 1986; Lanergan and Grasso 1984]. Reusing reliable components which have been thoroughly tested over time also reduces the potential for errors. The time required for developing new applications can be shortened by reusing existing software components which eliminates a portion of the normal development process [Cavaliere and Archambeault 1983].

The largest user of software systems, the U. S. Department of Defense (DOD), has recognized the importance of software component reuse in two recent initiatives, the development of the Ada<sup>®</sup> language and the Software Technology for Adaptable, Reliable Systems (STARS) program [Batz et al. 1983; Boehm and Standish 1983; Druffel et al. 1983; Horowitz and Munson 1984; Litvintchouk and Matsumoto 1984; Mathis 1986; Rauch-Hindin 1983; Wegner 1983, 1984]. The STARS program is a major DOD effort to improve software embedded in mission-critical systems by improving software practice. A primary objective within the application-specific area of the STARS program is the development of a reusable software technology [Batz et al.

---

<sup>®</sup> Ada is a registered trademark of the U. S. Government, Ada Joint Program Office (AJPO)



1983; Druffel et al. 1983]. The first step listed by Batz within this objective is the development of "cataloging procedures needed for describing and warehousing software," for its subsequent reuse.

The second initiative, the Ada programming language, was developed as the standard language for all DOD embedded software systems. It incorporates several major features which support reusability including generic program units, separately compilable modules, and information hiding (the separation of interface specifications from the module body).

### *1.1.2 Types of Reuse*

Software component reuse can occur in many different forms. The simplest and most familiar form of software reuse is the purchase of existing commercial packages like word processors, data base management systems, networking systems, and small business applications. These software products are often reused thousands of times (purchased by many customers). Software development environments and tools such as editors, compilers, and debuggers offer a similar example of reuse.

Subroutine libraries provide a different form of reuse. These collections provide a set of modules for a small application area where each module has a precise purpose and is fixed except for a few parameters which affect its operation in a well defined manner. The numerical analysis library from IMSL [1984] is one example of this form of reuse. Other forms of software reuse closely related to subroutine libraries are operating system service calls and built-in language subroutines such as sine, string handling functions, and square root.

Products from the software development process other than code can also be reused. Since coding is only 10 to 15 percent of the total development cost [Ramamoorthy et al. 1984], reuse of requirements, specifications, or design products can result in even larger gains in productivity. The Draco system [Neighbors 1984] is an example of the reuse of analysis and design components from a particular application domain. After analyzing a new domain, a Draco domain analyst creates a "domain language" which describes the objects and operations within the application area. A system analyst can then specify a "program" to Draco using this new domain language and via a transformation process (requires human guidance), Draco will produce executable code. Each time Draco converts a domain language program into executable code, the analysis and design information for the application domain is reused. The reuse of domain analysis and software development knowledge has given rise to several other forms of reuse such as application generators [Horowitz and Munson 1984], prototyping environments [Ramamoorthy et al. 1984], knowledge based programming paradigms [Waters 1985a, 1985b], and automatic programming systems based on formal specifications and transformations [Kaiser and Garlan 1987; Balzer et al. 1983; Topping and Baumel 1985].

### *1.1.3 Current Status*

It can be seen from the discussion above that reusability covers a wide spectrum of schemes and techniques ranging from simple library collections to application generators to knowledge based approaches. Terms such as "off the shelf software", "parts-based programming", and "software factory" have become associated with the concept of reusable software. Reuse has been

recognized as one of the major productivity techniques for the 1990's [Harrison 1986].

Several reuse projects such as ReadyCode at Raytheon [Lanergan and Grasso 1983, 1984], the General Purpose Component system at AT&T [Afshar 1985], and the Reusable Code system at Hartford Insurance [Cavaliere and Archambeault 1983] have had limited success. The reported successes, however, are only small-scale applications of reuse technology within limited application domains. Although useful for concept verification, they do not provide the essential elements necessary for the large-scale application of software component reusability.

One of the critical elements required for large-scale software reuse is a retrieval mechanism, a technique for locating the desired software component if it exists [Batz et al. 1983; Chandrasekaran and Perriens 1983; Curtis 1983; Freeman 1983; Grabow et al. 1984; Harrison 1986; Horowitz and Munson 1984; Jones et al. 1985; Lubars 1986; Mathis 1986; Ramamoorthy et al. 1986; Rauch-Hindin 1983; Standish 1984]. Reuse will only be valuable if the effort to describe, locate and reuse an item is less than the effort to create a new one. Current systems provide only minimal retrieval assistance to the developer or no assistance at all. Locating the software component to be reused is often a manual process assisted only by a printed catalog.

A major study conducted for the Navy by Hughes Aircraft Corporation surveyed 19 state-of-the-art software development methodologies concerning software component reuse and concluded that an effective retrieval mechanism was a key factor missing in every system examined [Grabow et al. 1984].

None of the methodologies used in large-scale development efforts provided a reliable way of storing and retrieving items . . . . . the retrieval of the correct item from the library was a manual process. (which was often so difficult that it was easier to code a new item than look for one to reuse.) . . . . the crux of the problem is our inability to precisely describe what a software component does.

Classification, the ability to organize and describe items in a collection, is the core of any effective retrieval technique. Current reuse projects have incorporated static classification schemes which are simple to create for small collections, but difficult to expand as new items are added to the collection (this is referred to as the "re-classification problem"). Static classification schemes provide only minimal descriptive capabilities and are usually limited to keyword retrieval techniques, frequently implemented in the form of printed catalogs. Using these schemes, the retrieval of a particular item from anything other than a very small collection becomes a laborious, frustrating task. The software classification methodology developed as part of this research addresses these deficiencies by providing a flexible schedule which is easy to expand and a rich descriptive capability which can be customized to the target domain.

## **1.2 Research Overview**

The objective of the research described herein was the design and development of a classification methodology and a retrieval model to support software component reuse. The primary motivation for this research was to improve reuse technology, thus reducing the cost of software development, especially for large-scale applications. The focus of an effective retrieval model is the capability it provides for describing

(selecting) the desired set of items. As part of this research, a faceted classification methodology was developed to provide this descriptive capability for software components. The resulting retrieval model is based on a consolidation of three research areas: (1) software component reuse, (2) information retrieval theory, and (3) classification schemes and techniques. It extends previous research in the area of software reuse by providing a retrieval mechanism for locating a desired software component within a large collection of items. The necessary cataloging and description capabilities are supported by a faceted classification scheme which is easy to use, extremely adaptable to changes and growth in the target collection, and concise in its descriptive format. It is a major improvement over existing keyword-based retrieval models and does not suffer from the expansion (re-classification) problems inherent in many other classification schemes.

Software component reuse is generally defined to be the use of source or object code developed for one application in a different application [Grabow et al. 1984]. This research extends the definition to include the reuse of other software products produced during the software life cycle such as requirements, specifications, designs, or test plans. This set of reusable products will hereafter be referred to as reusable software components. The approach taken was to study the problem of software component reuse and survey the domain of classification theory to identify feasible classification schemes that could be or have been applied to software. A faceted classification scheme for software components and a formal retrieval model were developed. The retrieval model was implemented as a rapid prototype using a data base management system modified to include the retrieval mechanism as a front end. The rapid

prototype supported the retrieval of code fragments from a collection of data structure algorithms. A subset of the International Mathematics Scientific Library collection was then added to the original software collection to validate the classification methodology in a different domain. The implementation and operation of the rapid prototype were then analyzed to determine the feasibility and utility of the retrieval model and faceted classification methodology in supporting the retrieval of reusable software components.

### **1.3 Overview of Following Sections**

Section 2 surveys the relevant research in software reusability, the dominant information retrieval models, and classification schemes and techniques, in particular, classification in library science and existing schemes used in software classification. Section 3 discusses the objectives and goals of a methodology for classifying and describing software components. A faceted software classification methodology is presented with the results of its application in a test domain. In Section 4, the design of a formal software retrieval model is described along with its implementation as a rapid prototype. Section 5 examines the addition of new components to the faceted classification scheme. Experiences and conclusions from the development of the faceted classification methodology and retrieval model and suggestions for future work are presented in Section 6.

## 2. REVIEW OF PREVIOUS WORK

Reuse, with its significant economic incentives, has become a major component in many current efforts to improve software productivity. Analysis has shown [Boehm 1981; Standish 1984] that the cost of software is usually an exponential function of its size; halving the size of the software which must be created (by reuse) would therefore reduce the cost by much more than half. Standish [1984] appropriately stated the motivation behind reuse with a modified quote from Bertrand Russell, "Software reuse has the same advantage as theft over honest toil." A survey of the literature revealed three research areas related to the development of a software classification methodology and retrieval model to support software component reuse: (1) software reusability, (2) information retrieval theory, and (3) classification schemes and techniques. The major contributions in each of these areas will be addressed as they relate to the research reported herein.

### 2.1 Software Reusability

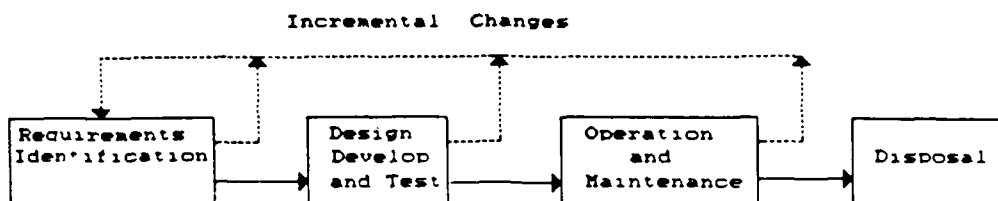
It has been recognized for some time that a fundamental weakness in the software development process is its focus on creation. Each new software system is constructed from scratch as if no other system like it had ever been developed. The study on Reusable Software Implementation Technology conducted by Hughes Aircraft Corporation for the U. S. Navy [Grabow et al. 1984; Huang 1985] found this to be true in a study of 19 current software development methodologies which covered a cross section of research, commercial, and industrial methods. The first two conclusions reported from this study were:

1. No credible methodology was examined which purported to provide the reuse of source code between dissimilar applications areas. In fact, where source-code level reuse occurs at all, it happens within fairly narrow application areas.

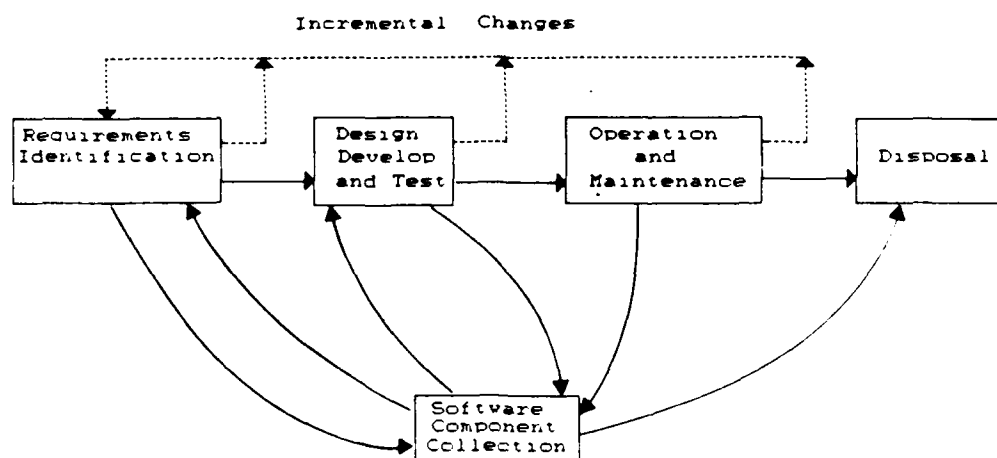
2. None of the methodologies used in large-scale development efforts provide a reliable way of storing and retrieving items from a code-level library. Some methodologies were able to implement libraries, but the retrieval of the correct item from the library was a manual process.

This is obviously an undesirable situation since studies have shown that a large percentage of applications contain common functions [Jones 1984; Goodell 1983; Lanergan and Grasso 1983, 1984]. While software development methodologies do not support reuse, cognitive research has shown that reuse (the use of analogies) plays an important role in the process actually used by software developers [Silverman 1985; Soloway and Ehrlich 1983, 1984]. When confronted with a new problem, most developers consider how solutions to previous applications might be used. This includes not only solutions from the developer's past experience, but also information from documents, plans, and algorithms taken from each phase of the software life cycle. Silverman [1985] suggests that "as soon as a solution (even a bad one) exists to a new problem domain, it is human nature to reuse that solution", and proposes a revision to the traditional software life cycle model shown in Figure 1(a). This revision, Figure 1(b), acknowledges reuse of components ranging from requirements to code fragments and suggests development of tools and techniques to support reuse.





(a)



(b)

**Figure 1.** Software life cycle: (a) traditional model and (b) reuse model

As described in Section 1, reusing software can imply the repetitive execution of application programs, the repetitive use of software development tools, the reuse of code fragments or subroutines from functional libraries, the reuse of components in a "software factory" setting, the reuse of analysis and design knowledge in transformation paradigms, or even the simple reuse of spreadsheet templates. This variety has created confusion concerning the meaning of "software reuse".

The type of reuse pertinent to the research reported herein involves the use of a software component in a situation other than the one for which it was originally created and with less effort than would be required to create a new component. The repetitive execution of application programs or software development tools is not included as their use does not usually require sophisticated retrieval support. Previous research falling under this definition of software reuse includes reusable code, design reuse, and transformation systems. Each of these areas builds on work from the preceding domains making the boundaries somewhat indistinct. The major research contributions in each area are discussed below with particular emphasis on the associated classification and retrieval aspects.

### *2.1.1 Reusable Code*

Reusing code fragments or subroutines is the "typical" view of software reuse. It is one of the oldest software engineering concepts with origins that can be traced back to the use of assembly language macros [Chandersekaran and Perriens 1983]. The concept of a reusable software component industry is almost two decades old. McIlroy [1976] proposed such

a software industry at the 1968 NATO Software Engineering Conference. Manufacturing facilities would produce and distribute standard interchangeable software components in a manner similar to the production and distribution of hardware components. This idea is still attractive and frequently mentioned as a viable option [Standish 1984; Wegner 1984]. Past research in code reuse has been concentrated in the areas of libraries and software factories.

#### *2.1.1.1 Functional Collections*

Subroutine libraries have been the most popular form of code reuse and are the area of software reuse most closely related to this research. Often referred to as functional collections, such libraries usually consist of functionally similar components (subroutines or functions) designed for the solution of very specific problems within a limited application domain. The most successful collections have been in the field of scientific programming such as the International Mathematics Scientific Library [IMSL 1984], a collection of Fortran subroutines for common mathematical and statistical problems. Typical of functional collections, each IMSL subroutine has a precise purpose and is fixed (i.e., not meant to be modified) except for a few parameters which affect the operation in a predetermined manner. This "black box" approach promotes using the subroutine without knowledge of the internal algorithms or design/performance information. Reuse is restricted to a function or subroutine call.

A printed reference manual is the only mechanism available for locating a desired subroutine in the IMSL collection and the cumbersome classification scheme provided can force even knowledgeable IMSL users to

spend hours searching for the correct routine. The IMSL reference manual groups the subroutines by functional areas. These groupings are then listed alphabetically as follows:

- A - Analysis of variance
- B - Basic statistics
- C - Categorized data analysis
- D - Differential equations
- E - Eigensystem analysis
- F - Forecasting
- G - Generation and testing of random numbers
- I - Interpolation
- L - Linear algebraic equations
- M - Mathematical and statistical special functions
- N - Nonparametric statistics
- O - Observation structure
- R - Regression analysis
- S - Sampling
- U - Utility function
- V - Vector-matrix arithmetic
- Z - Zeros and extrema

Although a fundamental guideline for classification schemes is to keep related topics close together (collocated), the IMSL organization intermingles statistical, mathematical, and other groupings. A user selecting the wrong cluster name (keyword) will be misled. An example from Curtis [1983] illustrates this organization problem.

If one wanted to perform a discriminate analysis, one would find little help looking under the D's. One must have enough statistical knowledge to know that discriminate analysis is a multivariate statistical technique. So, look under the M's, right? Wrong, the M's are devoted to Mathematical and Statistical Special Functions. In reading down the list, one must recognize that multivariate statistics provide an analysis of the structure underlying a set of observations. Therefore, for discriminate analysis one must go to the O's. One finds there a procedure for "multivariate normal linear discriminate analysis among several known groups" with the marvelously mnemonic title of ODNORM.

Using printed reference manuals for locating desired items is the standard technique in functional collections. The Statistical Package for the Social Sciences [SPSS 1984] and the UNIX Programmers Workbench [Kernighan 1984] are examples of other popular functional collections limited to simple reference manual type retrieval.

A recent innovation in the area of functional collections is a library of "objects" to support the object oriented methodology of software development [Booch 1986]. These objects (abstract data types) represent both the data structure of the entity involved and the actions which can be performed on that entity. A library organization for abstract data types (ADTs) has been developed at Brigham Young University based on a "knowledge structure" of is-a relationships [Woodfield 1985; Embley and Woodfield 1987]. The objects in the library are implemented as Ada packages. Each ADT is a pair (ADT-descriptor, ADT-implementation-set) where an ADT-descriptor describes the ADT and an ADT-implementation-set is a set of one or more Ada packages implementing the ADT. The developers, however, found the design of suitable descriptors for the ADTs based on this organization a more difficult problem than was originally envisioned. The object oriented library currently supports only two forms of cataloging and retrieval. The first, search by name, is not expected to be of much help to users since the names chosen by the library-administrator are not widely known or easily remembered. The second technique uses keywords to locate candidate ADTs. Once an appropriate ADT is found, users can browse other related ADTs via the is-a structure to locate the best ADT for reuse.

Functional collections have been a successful form of reuse. One reason for their success is that they concentrate on a small domain of well defined problems. Each collection is also usually limited to routines written in a single programming language. Reference manuals are typically the only retrieval mechanism available for locating components within these collections. Reference manuals can be adequate for locating items in very small collections, but retrieval becomes a problem as libraries grow in size. The successful examples of reuse in functional collections did inspire the investigation of reusable code libraries in industry.

#### *2.1.1.2 Libraries in Industry*

The economic incentives favoring reuse and the success of functional collections suggested similar applications in industry. The majority of these projects have been based on the functional collection model with a limited library of routines concentrated in a small domain. They do, however, demonstrate that the reuse of code in industrial environments can result in significant productivity improvements.

The ReadyCode system at Raytheon's Missile Systems Division was one of the first and most successful of these projects [Grabow et al. 1984; Horowitz and Munson 1984; Lanergan and Grasso 1983, 1984; Rauch-Hindin 1983; Jones 1984]. The project targeted the information processing systems at Raytheon comprised mostly of common business applications written in Cobol. The ReadyCode research team examined over 5000 existing Cobol programs and identified three major application categories: editing and selection programs (29%), updating programs (26%), and report programs (45%). A small subset of representative programs was then selected and

studied in closer detail. The research team concluded that over 40 percent of the code in these programs was redundant and could be standardized. Three prototype Cobol logic structures (select, update, and report) were developed and placed in production programming environments for testing. The positive results from this study were used to implement the ReadyCode system.

The ReadyCode library contains two types of reusable components, Cobol source code modules and Cobol logic structures, which can be combined to produce business data processing programs. The Cobol source code modules implement common business routines such as date aging, part number validation, and tax calculations and are used to complete the logic structures. The Cobol logic structures are skeletons (templates) of programs with prewritten identification divisions, environment divisions, data divisions, and procedure divisions. Logic structures are not complete programs (some parts are missing or incomplete) but serve as basic outlines for the intended application. Seven basic logic structures were initially identified and then augmented with various options (for example, one version with an imbedded sort and one without).

The ReadyCode system has been used to develop over 50 system applications at three different Raytheon plants with an average of 60 percent reusable code [Lanergan and Grasso 1983, 1984]. Programmers found that using well-documented routines with consistent styles resulted in programs that were easier for everyone to read and understand, thus reducing the maintenance effort. Overall, the ReadyCode system has produced a 50 percent increase in software productivity. ReadyCode

demonstrated the significant productivity gains and reduced maintenance costs available via the simple application of basic functional collection reuse technology, albeit on a small scale. As with functional collections, however, location of a desired component in the ReadyCode library is limited to either a keyword search or manual inspection of a module directory.

Many other reusable library projects have followed Raytheon's successful pattern. The Reusable Code system at Hartford Insurance [Cavaliere and Archambeault 1983] is another library of Cobol business applications containing program skeletons, algorithm templates, and source code modules. NASA-Ames has implemented a reusable library containing mostly Fortran modules [Jones et al. 1985] and the GPC system developed at AT&T Information Systems supports a library of reusable modules written in C [Afshar 1985]. The somewhat unique reuse library at Electricite de France in France contains a collection of abstract data types (packages) for scientific programming written in Fortran [Meyer 1982], while the Restructured Naval Tactical Data System (RNTDS) library contains modules written in the CMS-2Y language for reuse during the construction of real-time embedded shipboard software [Grabow et al. 1984]. Although each of these libraries varies somewhat in purpose and structure, there is one common characteristic which limits their size and reuse potential. Components within each of these libraries must either be selected manually from a printed catalog or by browsing an online directory using keywords. The retrieval techniques and static organization of these libraries were ported unchanged from the functional collection model. The problem of software classification was avoided by keeping the collections small.



In one application with a larger library, the catalog has evolved to overshadow the reuse library it supports. This catalog, the Bechtel Catalog of Computer Programs (BECCAT), is used at the Bechtel Corporation for cataloging software related to nuclear engineering design applications [Dumlao and Cook 1983]. Software in the catalog is indexed by subject, acronym, title, and host hardware. BECCAT is maintained by three professional librarians in the Bechtel Data Processing Library office. The average monthly cost for maintaining the catalog is over \$5000 and the printing cost for one edition of the catalog in 1982 was over \$6000.

#### *2.1.1.3 Software Factories*

The next development in code reuse was the Japanese software factory [Jones et al. 1985; Jones 1984; Matsumoto 1984; Tajima and Matsubara 1984]. A software factory is usually a separate division within a company organized for large scale software production. Standardized development methodologies and extensive automated support are used to maximize productivity. Software reuse is given a high priority. More than just a reusable code library, the software factory reflects a total management commitment to software reuse. Software factories train their programmers to reuse software and management strategies encourage (demand) reuse. For example, changing productivity indicators such as lines of code to the number of modules reused is a common practice [Prieto-Diaz 1985].

The Toshiba software factory [Matsumoto 1983, 1984; Matsumoto et al. 1981] manufactures software for real-time process control. Production rates in excess of 20,000 lines of code per person-year have been achieved with a reported reuse rate of about 50 percent. An object oriented development

methodology is used at Toshiba based on the reuse of abstract data types in the form of Ada-like packages. Each package is documented by a Description for Reusers (DFR) using a notation similar to SADT which describes graphically the function and interfaces of the packages. The DFR includes requirements and design information as well as the source code. The software factory at Hitachi, Hitachi Software Engineering (HSK), is similar in overall operation [Tajima and Matsubara 1984]. Reusability, however, is supported by a library of standard templates for frequently used programming structures. An automated environment called Skips II helps developers customize a selected template. HSK programmers are assigned exercises every month which encourage use of the template library.

Software factories like Toshiba and HSK have been very successful. Their success, however, is derived as much from standardized methodologies and automated development environments as it is from software component reuse. Their reuse libraries are based on the simple functional collection model. Each library concentrates on a single application domain, supports only one programming language, and relies on either manual inspection or keyword type searches for locating a desired component.

#### *2.1.1.4 Programming Languages and Environments*

The demonstrated increase in productivity and quality made possible with the simple reuse technology discussed above has renewed research interest in software component reuse. Several recent initiatives in the areas of languages and software development environments have had reuse as a major emphasis. The design, development, and required use of Ada had a major impact on programming languages since its sponsor, the DOD, is one

of the largest users of software systems in the world. It was recognized at design time that certain language features could enhance software reusability and their inclusion in Ada was given a high priority [Ichbiah 1983; Druffel et al. 1983; Horowitz and Munson 1984; Mathis 1986; Rauch-Hindin 1983; Wegner 1983, 1984]. Some of Ada's mechanisms which support reusability include: (1) a variety of programming units such as subprograms, tasks, and packages; (2) the separation of interface specifications from the module body (information hiding); (3) generic program units; (4) separately compilable program units; and (5) strong typing [Gargaro and Pappas 1987]. An Ada package consists of a specification part which contains the input and output parameters, data types, structures and procedures used by the package and a package body part that contains the code to implement the functions (actions). This separation of visible specification part and hidden function part supports both object oriented (abstract data type) and black box reuse paradigms. Packages with the same specifications can be interchanged without affecting other parts of a software system.

Ada's generic facility is another key feature which supports reuse. A generic program unit in Ada is a parameterized template for generating software components [Ledgard 1981]. A generic template can be instantiated (copied and completed) for different data types by specifying appropriate actual parameters. This allows a family of reusable software components to be specified by a single generic definition. These features are reinforced by the standardized use of Ada as the language for all DOD embedded software systems.

Several recent research projects have verified the utility of Ada's reuse mechanisms. The High Level Simulator (HLS) project [Litvintchouk and Matsumoto 1984] is a prototype simulation system providing a catalog of reusable modeling components. The HLS project relied heavily on Ada's generic mechanism to create classes of modeling components. A library of reusable software metric primitives implemented in Ada was developed by Pollock to support the study of software complexity metrics [Pollock 1985; Pollock and Sheppard 1985]. The separate compilation, generic program units, and package features of Ada were each cited as attractive reusability mechanisms.

Other languages are also providing mechanisms to support reuse. Modula-2, developed by Niklaus Wirth [Rauch-Hindin 1983], provides information hiding, strong typing, and separate compilation facilities. Smalltalk [Deutsch 1983; Wegner 1984] is an example of a growing collection of languages modeled after the object oriented development paradigm. These languages are based on the concept of data abstraction and hierarchical inheritance which originated in the Simula language. Objects with operations on an internal state are the primary language component. Operations and states may be inherited from previously defined classes which allows functionality to be incrementally defined in terms of previous functionality. Every object is described by (is an instance of) some class. Classes are arranged in a hierarchy with the property that if class A is a subclass of class B, then all operations implemented in B are available in instances of A. This allows new classes that are similar to old classes to be implemented with an effort that is proportional to the degree of dissimilarity. Lisp flavors and Objective-C are examples of other languages

based on the class hierarchy concept [Ledbetter and Cox 1985]. Experimental languages such as CONCISE [Gladney 1983] and BB/LX [Lenz et al. 1987] developed at the IBM Research Laboratory are extending language research in the areas of abstract data types and generic operators .

Work on reuse features in languages quickly migrated into development environment related areas. Environments can promote reuse by providing standardization, a common language interface, development tools and databases to assist reuse, and portability across hardware. The Ada Program Support Environment (APSE) and Kongsberg System Architecture (KOSAR) are examples of this approach [Wasserman and Freeman 1983; Wegner 1984; Pedersen 1984]. Other development environments such as the RPDE [Harrison 1986], AXE [Oskarsson 1983], SoftwareBase [Yeh et al. 1984a, 1984b], and SPS [Boehm et al. 1984; Wartik and Penedo 1986] present individual approaches to software reuse. Many of these projects have noted the need for research concerning component classification and retrieval mechanisms.

### *2.1.2 Application Generators*

As noted in the previous subsections, code reuse can improve software productivity and quality. Approaches which reuse other elements of the software development process, however, offer potentially larger improvements since they represent a larger portion of the total development cost. Application generators seek to reuse the analysis and design products of a specific problem domain [Biggerstaff 1984]. The typical application generator provides a nonprocedural language interface designed for a nontechnical person to use, often in the form of menu selections. The

generator system then combines these input parameters with its internal domain knowledge to produce an executable program.

### *2.1.2.1 The Draco System*

The Draco system [Neighbors 1984] extends the typical application generator model described above. Draco uses a different problem-domain-specific-language to describe the objects and operations in each given problem domain. It contains mechanisms for defining new problem domains as special-purpose domain languages and for converting statements in these new languages into executable programs [Partsch and Steinbruggen 1983]. At the lowest level, Draco contains domains that represent executable computer languages (currently only Lisp is implemented). New domain languages are defined by describing their syntax in a Backus-Normal-Form like formalism and defining their semantics by mappings from statements in the new language to statements in a previously defined domain language. The user can then formulate a problem solution (program) in terms of the new domain language. The essential steps are shown in Figure 2. As currently implemented, the user must interact with Draco to guide the refinement of a domain language program into executable code.

Since the prototype system became operational in 1979, only 10 Draco domain languages have been constructed [Neighbors 1984]. The system designers conclude that developing a good domain analysis and language implementation is a harder problem than originally envisioned. For example, a domain analysis done for tactical display systems was over 100 pages long.

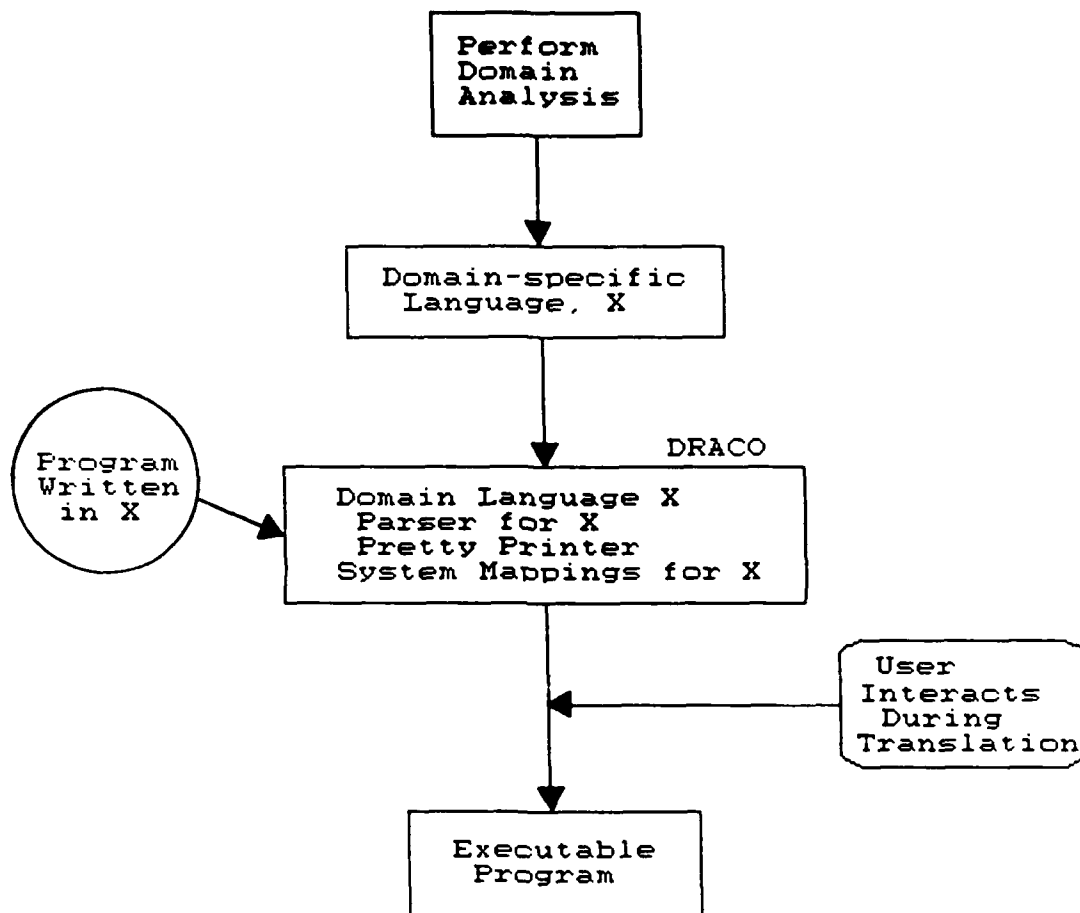


Figure 2. Draco System Overview

Several important questions remain unanswered about the practical use of the Draco approach. One is the efficiency of the final executable code since new domain languages are defined in terms of other domain languages. Another concern is the potential growth in domain languages which must be simultaneously mastered. Figure 3 shows a possible domain organization for a system which produces formatted statistical reports. The new statistics reporting domain language is defined in terms of 5 other domain languages. The designer of the new language must be familiar with the other 5 languages. Will this approach facilitate or hamper debugging and maintenance?

#### *2.1.2.2 KBMC*

The Knowledge-Based Model Construction (KBMC) system, which focuses on a single domain, represents the more typical approach to analysis and design reusability [Murray 1986]. KBMC supports discrete simulation modeling within the domain of queuing systems. No simulation language expertise is required of the user, who interacts with KBMC by describing the structure and components of the system to be modeled. The analysis and design information internal to KBMC is then "reused" to generate an executable simulation model in the simulation language SIMAN.

#### *2.1.3 Transformation Systems*

The boundary between application generators and transformation systems is sometimes difficult to determine. Biggerstaff [1984] uses the Draco system to demonstrate this problem, arguing that Draco could be classified in any one of three different categories: (1) a problem oriented



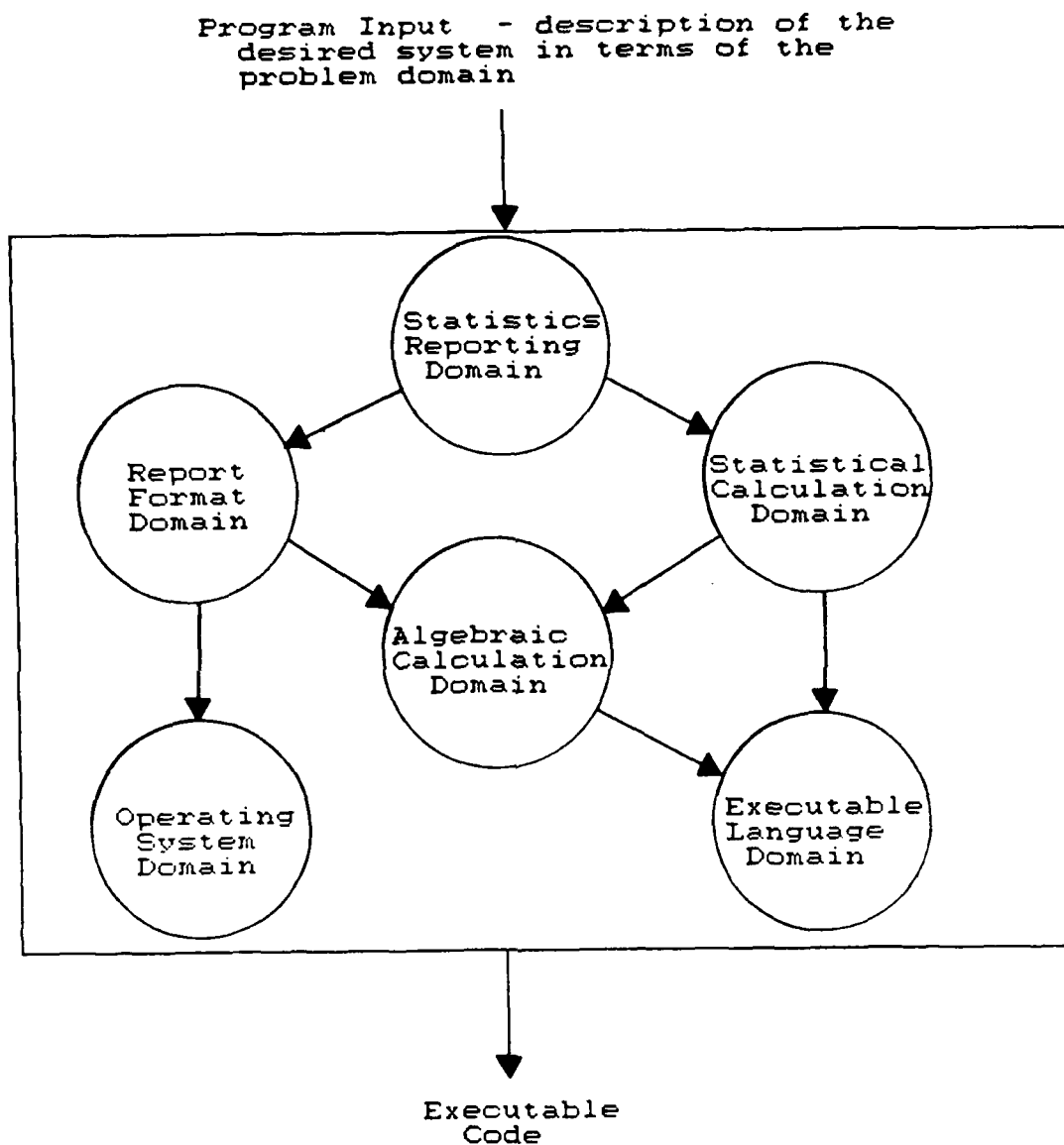


Figure 3. A Draco Statistics Reporting Domain

language, (2) an application generator, or (3) a transformation system. In general, a transformation system takes high-level specifications (in predicate calculus form, algebraic form, relational form, or declaration form) and transforms them into executable programs. The transformation process successively refines each abstract concept into a more and more concrete function until a final executable state is reached. A typical transformation system is pictured in Figure 4. The high-level problem specification is input to a composer which arranges the problem parts for internal system use and performs interface and consistency checks. The refinement process then translates the problem specification into the target language via an iterative transformation process. An optimizer works with the transformation module to eliminate inefficient implementation decisions.

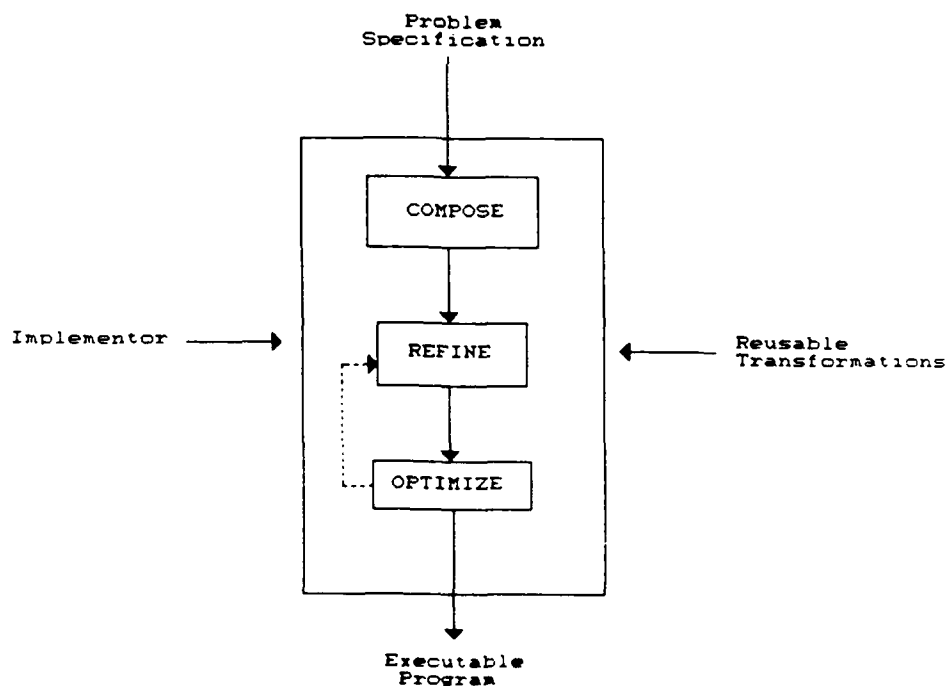


Figure 4. Transformation System Overview

Originally, transformation-based systems were presented as a new, "revolutionary paradigm" for software development [Balzer et al. 1983; Green et al. 1983]. This new paradigm would change the focus of software development and maintenance from the source code level to the more abstract specification level. Actually, this "revolutionary paradigm" is only a different form of software component reuse. The transformations house the reusable software components in these systems which can include analysis and design information as well as reusable code fragments or patterns [Horowitz and Munson 1984; Nourani and Jones 1985; Biggerstaff 1984]. Transformations are usually represented as either schema-based, pattern matching rules or as algorithmic procedures. A schematic rule contains an ordered pair of program units (schemes). When a match for the first scheme is detected, it is replaced by the second scheme in the rule with appropriate parameter substitution (analysis, design, or code reuse). An algorithmic transformation expresses the algorithm directly in a procedural language (code reuse).

The specification language (usually a wide-spectrum language) is a critical element of the transformation paradigm and is a limiting factor in current transformation technology [Partsch and Steinbruggen 1983]. It has been noted that many of the specification languages are more difficult to understand than the corresponding executable programs produced by the transformation system - a situation which does not improve software productivity or quality [Horowitz and Munson 1984]. GIST, a popular specification language used in several transformation systems, is one example. It is so complex and difficult to use that a paraphraser was developed to translate GIST specifications into a more understandable form

with disappointing results for the developers. With the paraphraser, several specifications "proven" correct and used for some time were found to contain serious defects [Balzer 1985].

Transformation systems, with one exception, currently exist only in research environments as experimental tools and are generally limited to a restricted range of small pedagogical problems. A representative selection of these systems is presented in the following sections.

### *2.13.1 PSI Program Synthesis System*

The PSI system, developed during the late 1970's at Stanford, is designed to generate an executable program from a formal description (GIST specification) of a problem [Green et al. 1979; Green 1976; Kant and Barstow 1981]. PSI is a modular transformation system composed of several integrated subsystems including a parser, an interface moderator, a program model builder, a code generator, and an optimizer. The program model builder (PMB) accepts partial program fragments as input and constructs a complete program model in an abstract modeling language [McCune 1977]. The incomplete program fragments are completed by the PMB through the use of internal procedural rules (reuse) or by querying the user for additional information. The completed program model is then transformed into a target language by the code generation module, PECOS.

PECOS [Barstow 1979] uses the abstract program model produced by the PMB and applies transformations that reuse design or coding information to produce an executable Lisp program. The transformation process consists of small step-by-step refinements of the program description

using a task oriented control structure. In each refinement cycle a task is selected and an appropriate pattern (transformation) is applied. A task may generate subtasks which are refined in a last-in, first-out sequence. When several transformations are applicable, the refinement is split with each transformation applied in a different branch of a tree of program descriptions. The root of the tree is the original abstract program model and the tree's leaves are programs in the target language. To keep the tree a reasonable size and produce efficient programs, the optimizer module, LIBRA, rejects certain possible refinement branches. LIBRA [Kant and Barstow 1981] contains reusable information about algorithm efficiency in the form of heuristics and analytic cost estimation which is used to prune the tree of partially refined programs and direct the order of further expansion. The PSI system has been successfully demonstrated on problems such as computing prime numbers and sorting linked lists but relies on human interaction to guide the transformation process.

The PSI research team has been responsible for several follow-on transformation projects [Smith et al. 1985]. Cordell Green and one PSI subgroup working at the Kestrel Institute used the technology in PECOS and LIBRA to develop the CHI system. The goals for CHI were to improve the transformation process (make it less dependent on human interaction) and develop a more usable specification language. The resulting specification language, V, has been transferred to commercial applications by Reasoning Systems, which markets a transformation-based software development system called REFINE. REFINE can be customized to specific problem domains by extending V via the addition of "knowledge packs" (domain specific transformations). REFINE has been selected by the

Lockheed Missiles & Space Company as the basis for EXPRESS, a research program to develop a transformation-based software development environment [Topping and Baumel 1985; Baumel et al. 1986].

A second major weakness of the PSI system was its dependence on human guidance during the refinement process. The development of Glitter, another offshoot of PSI, investigates methods for automating more of the refinement process by reducing the distance between the high-level specification and the final target language [Fickas 1985]. The work and experience from these projects provided the impetus for a long-range research plan, the Knowledge-Based Software Assistant (KBSA), underway at the Kestrel Institute [Green et al. 1983]. The KBSA plan outlines 5, 10, and 15 year research goals for improving automatic transformation system technology.

### *2.1.3.2 TAMPR*

The Transformation-Assisted Multiple Program Realization (TAMPR) system, developed at the Argonne National Laboratory, is an example of a special purpose transformation system [Boyle 1980; Boyle and Muralidharan 1984; Partsch and Steinbruggen 1983]. TAMPR was developed to adapt numerical algorithms to special problem or software environments via transformations. The numerical algorithms are represented as abstract prototype programs. Different sets of transformations called realization functions can then derive mathematical subroutines with the desired properties from the abstract representation. The transformations developed for TAMPR mainly focus on the transition of Basic Linear Algebra subroutines to in-line Fortran code.

The MENTOR transformation system [Nourani and Jones 1985; Partsch and Steinbruggen 1983] uses a processing model similar to TAMPR. A tree-manipulation language, MENTOL, makes transformations on algorithms represented as abstract syntax trees. The transformations are implemented as tree-rewriting operations.

### *2.1.3.3 Programmer's Apprentice*

The Programmer's Apprentice (PA), developed at MIT [Waters 1985a, 1985b; Rich and Waters 1983], consolidates reuse techniques from each of the areas presented above: a reusable library, reuse of analysis and design information, and reuse via transformations. The PA system provides five major functions: (1) a library which contains program plans (program fragments), (2) a plan editor for modifying program plans, (3) a coder which transforms a plan into Lisp code, (4) an analyzer which can construct a plan from a section of program code, and (5) a drawer to graphically display a plan.

A programmer uses the PA by selecting plans from the library and filling in unspecified portions with Lisp code or by modifying the existing code. A plan is composed of segments, each defined by input and output parameters and a set of specifications. Relationships between plan segments are described by "dependency links" which represent data or control flow.

The PA plan library is hierarchically organized from general abstractions to more specific implementations. For example, a plan about the concept of a loop could be specialized into an enumeration loop or a

search loop. No classification scheme or retrieval support is provided for the plan library. A user must specify the desired plan by name.

The research projects and industrial applications described above have demonstrated the enormous potential of software component reuse. However, in almost every study, whatever the form of reuse, a component classification scheme and retrieval mechanism have been cited as critical missing elements. Current techniques employ either printed reference manuals or simple keyword matching systems.

## **2.2 Information Retrieval**

Bartschi [1985] defines information retrieval (IR) as the process of searching for specific information stored among a great number of information items. Salton [Salton and McGill 1983] extends this definition to include the representation, storage, organization, and accessing of information items, where each information item consists of both the description that characterizes it and the actual information item. The overall objective of an IR system is to retrieve all the relevant items from a collection, while at the same time, retrieving as few of the non-relevant items as possible. This is in contrast to data base retrieval which selects only matching items. IR systems can vary significantly in surface characteristics, but are usually based on one of three underlying models: (1) boolean retrieval, (2) vector space algorithms, or (3) probabilistic methods.



### 2.2.1 Boolean Retrieval Systems

The boolean retrieval model, which offers simplicity and efficient operation, is used as the basis for most large commercial retrieval systems such as Lexis, Medlars, Dialog, BRS, Stairs, and the New York Times Information Bank [Croft and Ruggles 1983; Doszkocs 1983; Faloutsos 1985; Hafner 1981; Salton et al. 1983; Salton 1986]. The basic boolean model consists of an information item file and one or more directories (inverted files) of indexing terms (descriptors). For each descriptor in the inverted file, there is an associated list of information item numbers identifying each item which has been assigned to that descriptor. A simple boolean algebra can then be created by combining the descriptors with the set operators intersection, union, and negation. Boolean expressions in a query can then be evaluated by simply merging the lists of item addresses belonging to the descriptors in the query. The information items themselves are not involved in the query evaluation. For example, a query about the concept "information retrieval" might be stated as "INFORMATION AND RETRIEVAL". A boolean retrieval system would use the following procedure to identify the relevant information items:

1. Use the inverted index to retrieve the item numbers associated with the term INFORMATION. Call these item numbers Set 1.
2. Use the inverted index to retrieve the item numbers associated with the term RETRIEVAL. Call these item numbers Set 2.
3. Compute the intersection of Sets 1 and 2. Call these item numbers Set 3.
4. Use the information item file to retrieve the items identified by the reference numbers in Set 3.

A similar procedure would be used for the OR operator with union substituted for intersection in step 3. Set difference is usually used to implement negation.

The simplicity and efficiency of the boolean model are offset by several disadvantages [Salton et al. 1983; Biswas et al. 1985; Bartschi 1985]. First, the size of the output obtained is difficult to control. Depending on the assignment of terms to information items and the actual combinations of terms used in the query, either a large amount of output can be obtained or possibly, no items will be retrieved at all. Second, the retrieved items are not ranked in order of relative relevance. Each item retrieved is assumed to be as important as any of the other retrieved items. Third, there is no capability for assigning importance factors to the terms in the query or the descriptors in the inverted files.

These disadvantages derive from the retrieval of only items that fully match a given query. For example, if a query contains 10 descriptors joined by the AND operator, an item indexed by 9 of the 10 descriptors would not be retrieved. Similarly, combining the 10 descriptors with the OR operator would result in a large number of retrieved items, even items with only one of the requested descriptors would be retrieved. Bartschi [1985] describes this as the "all-or-nothing response". Items that do not contain all the index terms are treated exactly as items that contain none of the index terms. Recent work has introduced index term weighting, similarity measures, query term weighting, and fuzzy set theory as techniques for providing relevance measures in boolean retrieval systems [Bookstein 1980, 1981; Bartschi and Frei 1983; Bartschi 1985; Radecki 1981; McCune et al. 1985; Buell

and Kraft 1981a, 1981b]. A similarity measure, also called an association measure, is a technique for estimating the similarity between a given item and a particular query. Several systems discussed below incorporate these ranking techniques.

### 2.2.1.1 *Stairs*

The Storage and Information Retrieval System (Stairs) from IBM is somewhat unique in IR systems - no data bases are made available by its manufacturer, IBM. Most commercial IR systems provide large collections of information (e.g., Dialog, a simple boolean IR system, offers over 125 data bases) [Croft and Ruggles 1983; Salton and McGill 1983]. The Stairs user purchases only the IR software for use with private data bases. Stairs is based on the boolean retrieval model but can operate in two modes. The first mode, called SEARCH, is the basic boolean retrieval model described above (all-or-nothing) with no relevance ranking of retrieved items [Salton and McGill 1983]. The second mode of operation, RANK, uses term weighting of index terms to provide a ranked order of items. The value of a term associated with a particular information item is determined using the following formula:

$$\text{Value of term} = \frac{\text{ftd} \times \text{fts}}{\text{ndr}}$$

where ftd is the frequency of the term in the document, fts is the frequency of the term in the retrieved set, and ndr is the number of documents in the retrieved set containing the term. A final value is calculated for each

information item by summing the values of all terms which match query terms.

Stairs is considered to be a powerful, state-of the-art IR system. Recent large-scale tests conducted by Blair and Maron [1985] disappointed the IR community, revealing Stairs retrieved less than 20 percent of the items relevant to the searches. The average precision from the 40 test cases was about 75 percent, with recall averaging less than 20 percent. Recall and precision are generally defined by the following formulas:

$$\text{Recall} = \frac{\text{Number of relevant and retrieved}}{\text{Total number relevant}}$$

$$\text{Precision} = \frac{\text{Number of relevant and retrieved}}{\text{Total number retrieved}}$$

Experiments with several other commercial IR systems [Salton 1986] have shown similar results and are used to argue for the development of better retrieval models.

#### 2.2.1.2 *Liars*

Liars, the Louisiana Information Access and Retrieval System, was developed at the Louisiana State University to serve as a testbed for studying different schemes relating to fuzzy set retrieval techniques [Buell and Kraft 1981a, 1981b; Kraft and Buell 1983]. Liars contains two major components; a high-level user interface which retrieves items from the

collection by invoking the second component, a database management system (Lorelei) extended to support IR requirements. Boolean queries are input to the system with "weights" optionally attached to each term. A mathematical module then accesses the inverted files and computes the retrieval status value for each relevant information item, ranking them for final output. The separation of major functions into individual modules supports experimentation using different retrieval strategies. Changing the fuzzy set model in the mathematical module does not affect other components in the system. It is interesting to note that Liars and Lorelei were both written in interpreted Business Basic.

### 2.2.1.3 *Fakyr*

Fakyr is another IR research system, but on a much larger scale than Liars. Developed at the Technische University in Berlin, Fakyr is a boolean retrieval system supporting ranked retrieval, fuzzy retrieval, retrieval by classes, relevance feedback, and a variety of clustering techniques [Bollman et al. 1983]. Fakyr represents 45 man years of effort and consists of over 130,000 lines of PL/1 code and 30 assembler subroutines.

Fakyr has been used to test over 40 association measures and fuzzy retrieval strategies with some enlightening results. One association measure, the E. Konrad and H. Zuse, sometimes produced better results than the other techniques. It was not revealed until the end of the experiment that the E. Konrad and H. Zuse measure was actually a disguised random function. Experiments involving fuzzy set based retrieval techniques concluded that fuzzy set retrieval is no better than the simpler term weighting technique for relevance ranking. Preliminary results from work in progress indicate

that clustering techniques can improve retrieval performance and that certain hypotheses accepted in the past need to be re-examined (Salton's cosine measure and the measure of Tanimoto may not be the best association measures).

### *2.2.2 Vector Space Retrieval*

The vector space retrieval model was developed in response to the disadvantages of boolean retrieval (no ranking of retrieved items, no control of output size, and no importance weighting of query terms), and is based on the concept that items with many descriptors in common with the query are probably more relevant to the user's request. In the vector model, both information items and search requests are expressed as sets of weighted terms (attribute vectors) of the form  $I = (i_1, i_2, \dots, i_n)$  and  $Q = (q_1, q_2, \dots, q_n)$  where  $i_b$  and  $q_b$  represent the weight (importance) of term  $b$  in item  $I$  and query  $Q$  respectively [Salton 1982; Bookstein 1983]. With both queries and information items represented as attribute vectors, similarity measures can be computed between a given query and each stored item to produce a ranked output of relevant items [Radecki 1983].

The vector space model provides importance weighting of index and query terms, ranking of the selected output according to relevance, and some control of output size using threshold values to limit the number of items presented to the user. However, selection of a particular similarity measure appears to be completely arbitrary and none have been shown to be consistently superior. Several examples have been documented [Radecki 1983] where existing similarity measures actually produce erroneous results. These disadvantages coupled with the computational overhead of

maintaining frequency counts for each index term in each information item in the collection have kept vector space IR systems limited to relatively small collections in research environments.

#### *2.2.2.1 Smart*

The Smart IR system was developed at Harvard in the 1960's as an experimental tool for the evaluation of vector space retrieval concepts [Salton 1971]. Smart is based on the classic vector space retrieval model described above with the cosine of the angle between the item vectors and a query vector used as the similarity measure. For two decades, Smart has served as a testbed for experiments with similarity measures, automatic indexing schemes, clustering techniques, and most recently with relevance feedback research [Salton 1981, 1982, 1986; Salton and Wu 1981; Salton and McGill 1983]. It is interesting that during this lengthy period of study, no large-scale comparison with a boolean system has been reported. One set of tests claiming a 40 percent improvement for vector space retrieval was reported by Salton [1986], but the item collections were too small for a valid evaluation.

#### *2.2.2.2 Sire*

The Syracuse Information Retrieval Experiment System (Sire) is a hybrid system combining concepts from both the boolean and vector space retrieval models [Salton et al. 1983; Salton and McGill 1983]. Retrieval in Sire is a two phase process. First, a boolean query is processed using the standard inverted file technique. The information items selected by this process are then ranked during the second phase using techniques from the vector space

model. Weights assigned to each term in the selected items are used as input to a cosine similarity measure for ranking the final output. Before the cosine measure is computed, the original query is "flattened". All original query operators are replaced by OR operators. The query (A AND B) OR (C AND D) would become A OR B OR C OR D (the query vector (A,B,C,D)). This illustrates another disadvantage of the vector space model; representing queries as attribute vectors destroys the boolean structure.

### 2.2.2.3 *Caliban*

Caliban is an experimental IR system developed at the Swiss Federal Institute of Technology (ETH) for use on small personal computers [Bartschi and Frei 1983; Bartschi 1985]. As part of the design effort to move away from boolean technology, the usual inverted file retrieval structure was replaced with a tree structure. Each fixed-sized node contains structural information, the name of the node, node type, and a link to the attached data itself. The actual data is stored as a separate linked list to accommodate variable-sized information items.

To retrieve information using Caliban, the user specifies a "virtual information item" (fills out a template describing the item). Weights may be assigned to query descriptors to express their relative importance. However, as in other IR systems based on fuzzy set logic, this eliminates the use of boolean operators for describing queries. To simulate the NOT operator, Caliban allows negative weights for query terms. A term can be excluded from a query by assigning it a high negative weight. The similarity measure is computed as the sum of the products of the query term weights and index term weights along the tree path to each matching information item



[Bartschi and Frei 1983]. To improve recall, item clustering can be simulated by including entire subtrees in the retrieval response.

### *2.2.3 Probabilistic Information Retrieval*

The third and newest direction of IR research is on probabilistic retrieval models. In the probabilistic model, items are selected whose probability of relevance to a query is largest [Bookstein 1983; Croft and Ruggles 1983; Maron 1983; Salton 1986]. The probability of an item's relevance to a particular query is determined by  $P(\text{relevance/item})$ . What characteristics and mechanisms can be used to determine this probability have not been discovered.

Current experimental systems (only a handful exist) use simple characteristics such as term frequency or expected retrieval costs as estimates of relevance probability [Van Rijsbergen 1979]. The term frequency approach is based on the assumption that index term distribution over the relevant items is different from their distribution over the non-relevant items [Robertson et al. 1981; Bookstein 1983; Robertson et al. 1983]. Several years of research have produced very little progress in this area. As Rijsbergen states, "The probability ranking principle assumes that we can calculate  $P(\text{relevance/item})$ , . . . this is an extremely troublesome assumption . . . we do not know which are the relevant items, nor do we know how many there are so we have no way of calculating  $P(\text{relevance/item})$ ." Current prototype systems employ an estimating technique to determine the probability of relevance coupled with an iterative relevance feedback mechanism to improve the estimate [Bartschi 1985; Maron 1983]. Bartschi also describes another major problem with the probability model, the "curse of

dimensionality". The large number of descriptor terms for each information item in a large collection makes the calculation of actual probabilities impractical. Only a few prototype IR systems using the probabilistic model have been implemented - with rather disappointing results.

### *2.2.3.1 Harter's Model*

The probabilistic scheme based on the assumption that different term distributions exist over the sets of relevant and non-relevant items is called Harter's model. Experiments with a prototype system based on Harter's model have been conducted at the University of Cambridge [Robertson et al 1981]. Using a test collection of 11,000 items from the National Physical Laboratory, a 10 percent recall measure was achieved with precision at the 49 percent level. This poor performance was attributed to the "estimation" problem (how to calculate the probabilities).

### *2.2.3.2 University of Massachusetts System*

A probabilistic IR system has been developed for a collection of bibliographies, reference works, and technical reports for the Department of Computer and Information Science at the University of Massachusetts [Croft and Ruggles 1983]. Employing term frequencies and relevance feedback techniques, the retrieval model uses a four step process:

1. Initially rank the information items using term frequency counts (a combination of probabilistic and vector space techniques).
2. Obtain relevance judgments from the user for the top 10 ranked items.
3. Expand the query using a subset of the terms from the relevant items.
4. Rerank the collection using probability parameters estimated from step 2.

This system is actually a simple vector space IR model with a weak probability estimation scheme thrown in as a final step. It is obvious that a great deal of work remains to be done to make probabilistic retrieval a usable technique.

The retrieval systems and underlying models described above each have unique strengths and weaknesses. Boolean systems offer simple implementation methods offset by an "all-or-nothing" retrieval process. This approach is unsatisfactory for software component selection. Vector space models provide ordered retrieval based on an item's relevance to the query, but the similarity measures used in existing systems have not been validated. Different similarity measures are needed for software component selection and ranking. The probabilistic model remains ill-defined and unusable.

## **2.3 Classification Schemes and Techniques**

This section introduces the basic concepts of classification theory as derived from a review of library science literature. General classification terms and techniques are presented to provide the background for subsequent discussion of the major library classification schemes and research. Existing software classification schemes are surveyed concerning their potential for classifying software component collections for reuse.

### *2.3.1 General Classification Terms*

Classification is the process of distinguishing things or objects which possess a certain property or characteristic from those that lack it and grouping things or objects which have the property or characteristic in

common into a class or category [Chan 1981]. The word "class" is derived from "clasis", the calling together of Roman citizens in groups according to their degree of wealth [Buchanan 1979]. Classification displays the relationships between things and between classes of things providing a form of abstraction which allows generalization over a large body of knowledge.

Buchanan [1979] identifies two types of relationships between classes; syntactic and hierarchical. Syntactic relationships involve a grammatical connection between classes to represent a subject. These relationships may be simple (elemental), superimposed, compound, or complex. Simple relationships define one kind of thing only, such as "automobiles". Superimposed relationships are simple relationships with more than one characteristic. "Convertible automobile", for example, is a kind of vehicle defined both by form and type of top. Complex and compound relationships deal with more than one kind of thing. A complex relationship is like a chemical mixture, the elemental compounds are separable. A compound relationship is more like a chemical compound where the elements fuse and cannot be separated. The subject "comparing flight characteristics of bees and airplanes" is an example of a complex relationship, and "sight in bats" is a compound relationship.

The second type, hierarchical relationships, was originally derived from the fields of logic and philosophy, and can represent a natural relationship such as genus to species (e.g., animals - mammals) or a broader-narrower type of relationship where natural orderings may not exist (e.g., software metrics - validating software metrics). A hierarchical structure progresses from the general to the specific, dividing the subject into

successive stages of classes and subclasses. The characteristic used for dividing each class is called a facet [Chan 1981]. In Figure 5, the class literature is divided into three facets; language, form, and period.

In hierarchical relationships, a class which contains another is said to be superordinate, the contained class is subordinate to the containing classes. Coordinate classes share the same immediate superordinate class but are neither superordinate or subordinate to each other. Classes which are neither broader nor narrower than each other while not sharing the same immediate superordinate class in the same hierarchy are said to be collateral. In Figure 6, the class Functional is subordinate to Languages, superordinate to Lisp, APL, and SASL and coordinate with the class Imperative. The

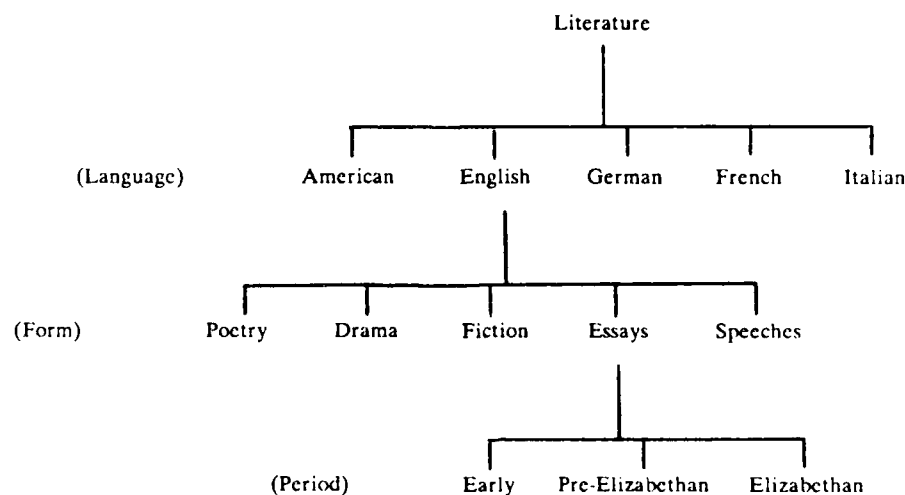
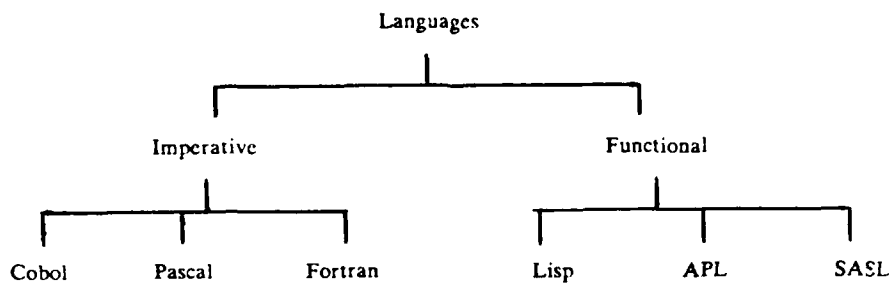


Figure 5. Literature Hierarchy



**Figure 6.** Computer Language Hierarchy

classes Cobol, Fortran, and Pascal are coordinate with each other, collateral with the classes Lisp, APL, and SASL and subordinate to the class Imperative. The coordinate elements on each level form an array (e.g., in Figure 5, American literature, English literature, German literature, etc.). A chain is a linked string of elements with each element representing a lower level of the hierarchy (e.g., Literature, English literature, English essays, Elizabethan essays).

There is not always a natural hierarchy of the facets in each class. For example, literature may be divided first by language and then by form, but could also be divided first by form and then by language. The order of the facets specified by the classification system is called the citation order. A classification schedule provides a list of all classes and their relationships in a prescribed order.

### *2.3.2 Types of Classification Systems*

There are two general types of classification schemes; enumerative and faceted. Enumerative schemes take a subject area and divide it into successively narrower classes listing all the elemental, superimposed, and compound classes arranged in order of their hierarchical relationships. This "listing" of all possible subjects is the major disadvantage of the enumerative scheme [Buchanan 1979]. Any subject (class or subclass) which does not appear in the schedule cannot be classified (located in the collection).

The second type of classification scheme, faceted, is more of a building block approach emphasizing subject analysis and synthesis [Chan 1981]. An analysis process is used to construct the classification schedule. Subjects to be classified are analyzed and divided into their elemental terms (e.g. things defined by only one characteristic). Only the elemental terms and their relationships are listed in the faceted schedule. Recurring divisions are not repeated in each major class as they are in the enumerated scheme. The elemental terms are listed separately for application to all subjects as needed. Synthesis is then used to express a superimposed, complex, or compound class by assembling its elemental parts from the facets according to the citation order. Since the analysis and synthesis process plays such a major role, faceted systems are also called *analytico - synthetic* classification in the literature [Chan 1981; Buchanan 1979; Vickery 1960]

Figures 7 and 8 illustrate the difference between a faceted scheme and an enumerative scheme for identical subject areas. Simple, superimposed, and compound classes are listed ready-made in the enumerative version,

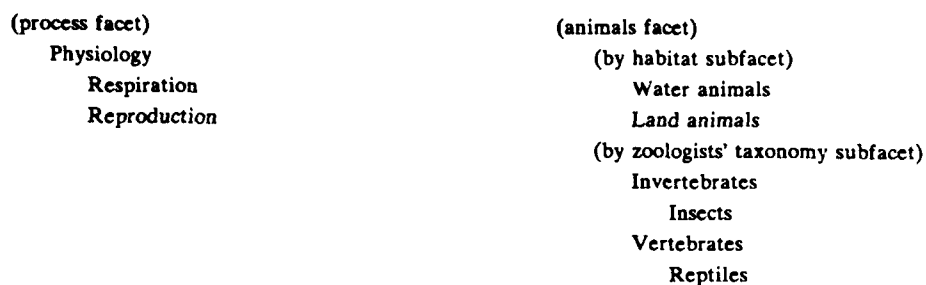


Figure 7. Faceted Classification Example

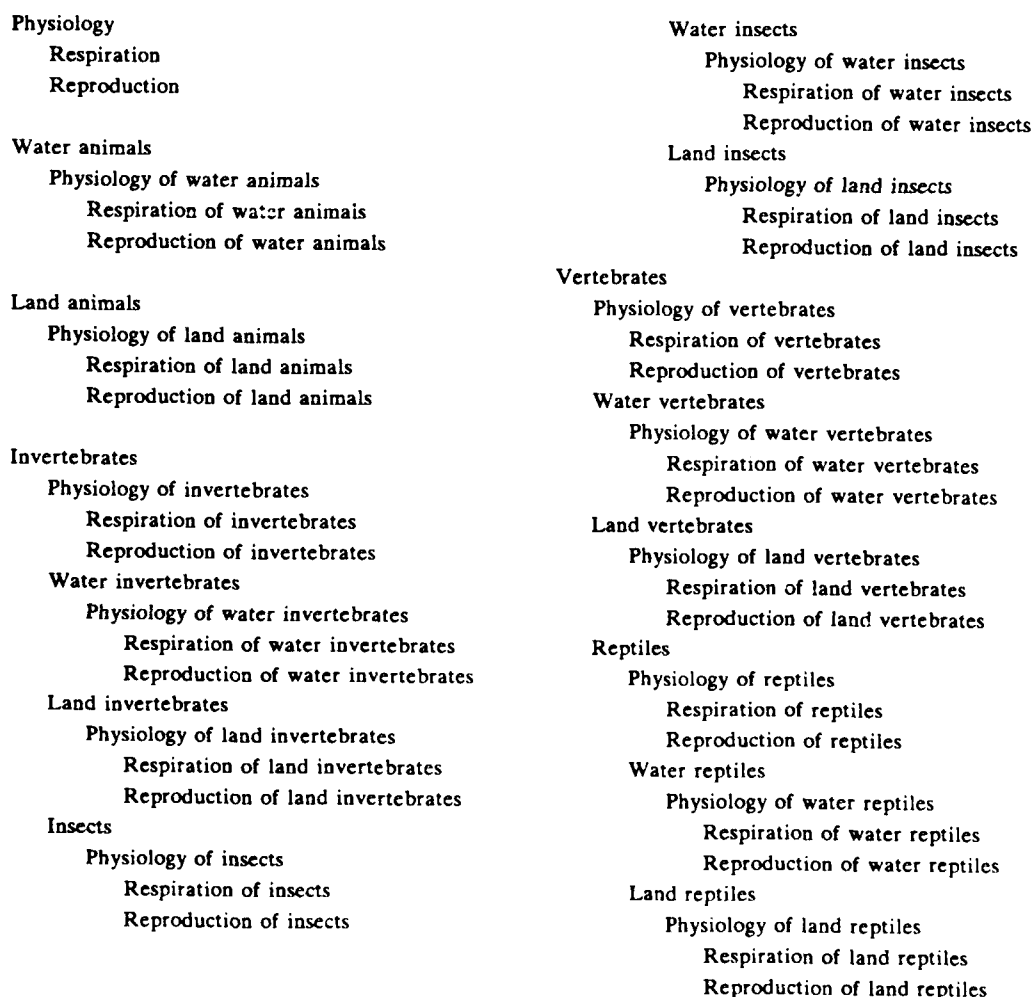


Figure 8. Enumerated Classification Example





Prior to the publication of DDC in 1876, most library classification schemes were based on philosophical hierarchies of knowledge and documents were numbered according to fixed locations on the shelves. Each book had a fixed location. This fixed location technique predominated in most libraries until the end of the nineteenth century [Taylor 1985].

### *2.3.3.1 Dewey Decimal Classification*

Melvil Dewey (1851-1931) was an assistant librarian at Amherst College when he developed the DDC. First published (anonymously) in 1876 as a pamphlet for library classification, Dewey's scheme introduced the ideas of relative location and relative indexing as an alternative to the fixed location arrangement [Chan 1981; Taylor 1985; Foskett 1982]. Using relative location (putting the numbers on the books instead of on the shelves), a new book could be inserted into the middle of an existing sequence instead of always being added to the end (the usual practice for fixed arrangements).

The DDC is the most widely used library classification system in the world, and is used in over 85 percent of the libraries in the United States and Canada [Michael 1976]. Dewey based the DDC in part on an earlier scheme devised by W. T. Harris who based his scheme on an inverted order of Francis Bacon's classification of knowledge [Bloomberg and Weber 1976; Comaromi 1976; Chan 1981; Taylor 1985]. Dewey arranged his scheme into 10 main classes with each class divided into 10 divisions and each division into 10 sections [Osborn 1982; Bloomberg and Weber 1976]. The 10 main classes are:

000	Generalities
100	Philosophy and related disciplines

200	Religion
300	Social sciences
400	Language
500	Pure science
600	Technology (applied sciences)
700	The arts
800	Literature
900	General geography and history

DDC is basically an enumerative scheme listing all elemental, superimposed, and compound classes. The nineteenth edition consists of three volumes listing over 21,000 topics [Foskett 1982]. It is interesting to note that recent editions of the DDC have included limited means of expansion by synthesis in some sections.

DDC is not a classification by subject but by discipline. A given subject may appear in any number of disciplines with the various aspects of a subject being listed together in the relative index. This means the DDC scatters subjects, violating one of the fundamental functions of classification, namely the collocation of materials on the same subject and related subjects [Bloomberg and Weber 1976]. Figure 9 shows how subjects on different aspects of railroads are scattered throughout the DDC collection.

The DDC scheme also demonstrates another limitation of enumerative classification. The schedule has been subdivided into 1000 main sections numbered 000-999. There is no possibility of adding more. New sections, such as computer science or electrical engineering, are added at lower hierarchical levels. This has created some sections with very deep hierarchies while others remain quite shallow. For example, "Understanding the microprocessor" has a DDC code of 621.3819583, but in the logic division (160), none of the sections are subdivided and two are vacant [Osborn 1982].

<u>Aspects of Railroads</u>	<u>DDC Number</u>
Railroad building	625.1
Railroad economics	385.1
Model railroading	625.19
Railroad law	343.095
Government control of railroads	350.875
Railroad safety	614.863
Bibliography of railroads	016.385
Railroad station architecture	725.31
Fiction about railroads	808.839356

**Figure 9.** Subject Scattering in DDC

### 23.3.2 *Universal Decimal Classification*

The UDC was developed in 1894 by two Belgian lawyers, Paul Otlet and Henri LaFontaine, as a universal classification scheme for all published literature [Foskett 1982; Taylor 1985]. Their initial work was based on expanding the DDC (with Dewey's permission) and resulted in the First International Conference on Bibliography in 1895. After 10 more years of work, the *Manual de repertoire universal bibliographique* was published in 1905. The UDC was quickly adopted by many libraries in Europe where it is still the dominant scheme. Over the years, the development of UDC as a universal bibliography was abandoned, but its development and use as a library classification scheme continued.

UDC retained DDC's decimal notation and follows the basic outline of DDC in its 10 main classes, but to suit its universal purpose there are many more detailed subdivisions. The major contribution of UDC to

classification theory was its synthesis of complex subjects [Taylor 1985; Prieto-Diaz 1985]. UDC introduced auxiliary schedules and a special notation to connect and relate terms listed in these separate schedules. UDC was the first major classification scheme to use such a system of common facets and synthesis [Chan 1981]. This powerful feature is, however, also the source of a major weakness in UDC; its notational complexity. Table 1 shows the various symbols which can be used to form complex subjects. The overloading of the equal sign, colon, and parenthesis symbols with multiple meanings adds to this unnecessary complexity. For example, 61(038)=20=50 denotes "Italian - English dictionary of medicine". The synthesis comes from 61 medicine, 038 dictionary, 20 English, and 50 Italian. A second example from Foskett [1982] shows the length of the notation when using the colon to combine foci. The notation 669.3:621.785:621.643.2 represents the "annealing of copper pipes" (669.3 copper, 621.785 annealing, and 621.643.2 pipes - fluid distribution).

**Table 1.** Synthesis Indicators in UDC

Symbol	Meaning
+	Combining two separate numbers
/	Combining two or more consecutive numbers
:	Relationships between two subjects
::	Similar to the colon sign
=	Language
(0...)	Form
(1/9)	Place
(=0/9)	Race and Nationality
"..."	Time
A/Z	Alphabetical subarrangement
.00...	Point of view
=05...	Persons

The international aspect of UDC is both an advantage and disadvantage. The revision process, as in many international ventures, is slow and clumsy as it moves through a network of committees and experts. The schedules for 681.3 (computers) were compiled in the late 1950's and early 1960's, eventually being approved for publication in 1965. They have not been updated since.

Despite these problems, UDC is used by a large number of libraries, particularly in Europe, Latin America, and Japan. It has been selected as the official classification scheme for scientific and technical libraries in the Soviet Union and Eastern Europe. UDC has demonstrated the utility of faceted classification, especially for highly specific subject areas.

#### *2.3.3.3 Library of Congress*

The Library of Congress (LC), founded in 1800, originally classified its collection by size (folios, quarto, octavos, etc.). In 1812, a new system based on Benjamin Franklin's Library Company of Philadelphia was adopted. Two years later, British soldiers burned the Capitol destroying the entire 3000 book library. Thomas Jefferson offered to sell his library of 6000 books to Congress to replace the lost collection. Jefferson's classification system, which came with the books, was used by the LC until the end of the nineteenth century [Foskett 1982; Taylor 1985; Chan 1981].

When the library moved to a new building in 1899, Dr. Herbert Putman, the new Librarian, decided to reorganize and re-classify the entire collection. After studying the available classification schemes, the LC staff decided to design their own special scheme to fit the library's collections and

unique services. The typical classification approach based on the theory of knowledge mapping was not used. Instead, separate schedules were developed by specialists for each individual class based on the needs of the library. This technique is called *literary warrant* [Foskett 1982; Chan 1981]. The schedules, each containing an entire class or subclass, were developed and published separately. For this reason, the LC scheme is often called "a coordinated series of special classifications" [Chan 1981; Taylor 1985]. LC is essentially an enumerative scheme, and as would be expected, its schedules are voluminous covering more than 6000 pages in 37 separate volumes. Any subject not specifically listed in the schedules cannot be classified by LC.

LC consists of the following 21 major classes:

A	General works and polygraphy
B	Philosophy and religion
C	History: auxiliary sciences
D	History: general and old world
E-F	History: America
G	Geography; anthropology; folklore
H	Social sciences
J	Political science
K	Law
L	Education
M	Music
N	Fine arts
P	Language and literature
Q	Science
R	Medicine
S	Agriculture
T	Technology
U	Military science
V	Naval science
Z	Bibliography and library science

Each class contains its own index. There is no general index for the complete LC collection. This separation of schedules even extends to revisions. New editions of individual schedules are prepared and published as needed, independent of one another. The LC notation is mixed letters and numbers. Main classes are denoted by a capital letter, sections by a second capital letter, Arabic numerals (1 to 9999 with no decimals) are used for divisions, and Cutter numbers, if necessary, for individual books (e.g., Q Science, QA Mathematics, 76 Computer science). Decimal numbers were added later as a way of inserting new subjects where no room existed in the schedule (e.g., Q Science, QA Mathematics, 76 Computer science, .6 Software). LC notation does not enforce a hierarchical order, and recent editions of many schedules are adopting alphabetic arrangements. Foskett [1982] points out that the growing use of alphabetical arrangements in LC reflects a "mark and park" philosophy, abandoning the very purpose of classification. For example, all books related to computer software are grouped in class QA76.6 and ordered alphabetically by author name. No further subdivision of software, a very general subject, is provided in LC [Prieto-Diaz 1985]. An example of LC's lack of hierarchy is provided in the TX724 subclass [Chan 1981]. TX724 represents Jewish cookbooks. When a number was needed for Oriental cookbooks, the number TX724.5 was assigned. Oriental cookbooks is certainly not a subdivision of Jewish cookbooks.

LC has been adopted by many academic and research libraries. While Michael's survey [1976] found 85 percent of all libraries in the United States and Canada using DC, among libraries with holdings of more than 500,000 volumes, over 62 percent use LC. The two reasons given for use of LC were: (1) the orientation of LC toward research collections, and (2) the economic



advantage offered by the LC cataloging services. However, the lack of a predictable basis for subject analysis and no logical hierarchy make the use of LC in automated retrieval systems impractical.

#### *233.4 Bibliographic Classification*

BC is a mixture of enumerative and faceted schemes. Developed in 1908 by Henry Evelyn Bliss, BC has been more popular as a scholarly artifact than as a practical classification system. This is mainly due to the time span required for its completion. The outline for BC was published in 1908, but completion of the schedules took Bliss the rest of his life [Foskett 1982; Mills 1972]. Volume I which contained the common facets and classes A to G was not published until 1940. Volume II, with classes H to K, followed in 1946, and Volume III and the index were not published until 1953.

Bliss considered the order of basic classes as the most important part of a classification scheme and based his order of classes on three major principles: collocation of related subjects, subordination of special to general, and gradation by speciality [Bliss 1985]. However, Bliss found the consistent application of these principles difficult in practice. The following example from the Literature class shows a violation of the first principle, collocation.

YE	Elizabethan, Jacobean and Caroline periods
of English literature	
YEI	Poets
YEN	Drama - Shakespeare's contemporaries
YEP	Dramatists
YEW	Caroline period
YF	Shakespeare

In order to obtain a two letter base notation (YF) for a subject likely to have a large amount of literature (Bliss had a passion for short notation), Bliss ignored the collocation principle. General works of the Caroline period (YEW) are separated from individual authors of the same period (YEI and YEP), and Shakespeare (YF) is separated from his contemporaries (YEN).

BC supports two forms of synthesis; common facets which may be applied anywhere in the schedule, and private facets for special subject areas [Foskett 1982]. It is confusing that Bliss recognized the potential of synthesis and provided for common facets, but implemented BC mainly as an enumerated scheme. For example, Bliss provided a common facet C for indicating founder in the main class Religion (P), and then proceeded to enumerate Buddha, Mohammed, and Christ as PJC, PKC, and PNB respectively.

Bliss was a critic of the complex notation used in CC and UDC, and wanted a shorter scheme for BC. The main classes are denoted by capital letters and common subdivisions by the numerals 1 to 9. Place subdivisions are shown by lower-case letters with a comma used to indicate language and period divisions [Langridge 1973]. This creates a problem, however, since the notation for language and period are identical and both are indicated by using a comma. Bliss also did not provide any guidance for the filing order of the different symbols. He actually confused the issue by sometimes filing lower-case after upper-case (e.g., CIg after CIG), but in other situations reversing the order (e.g., HYe before HYA, not after HYE). BC has been adopted by only a handful of libraries, mostly in Great Britain.

### *2.3.3.5 Colon Classification*

Shiyali Ramamrita Ranganathan is considered by many to be the leading theorist in the field of classification, and his Colon Classification (CC) published in 1933 has been a significant influence on modern classification systems [Gopinath 1972; Chan 1981; Foskett 1982]. The scheme's name came from the role of the colon symbol (:) as the only facet indicator in the first edition of CC.

CC was the first completely faceted library classification scheme. Most subjects are compounds specified by linking together elemental terms from different facets (categories). In CC, all facets are related to one another in a fixed citation order expressed by Ranganathan as PMEST (Personality, Matter, Energy, Space, and Time) [Langridge 1973; Chan 1981]. Personality, Matter, and Energy are specific for each class and are first in the citation order. Time and Space are treated as common facets which can be applied to any division and follow PME in citation order. Subjects are composed from the elemental terms in different facets using a "facet formula". The general facet formula (overall citation order) is PMEST.

A major disadvantage of CC is its complex notation which combines arabic numerals, capital and lower-case letters, Greek letters, brackets, and several other punctuation marks [Foskett 1982]. The Generalia classes are represented by arabic numerals with the other main classes using mostly

capital letters mixed with a few Greek letters. Basic elements within each class are shown by arabic numerals as illustrated below.

L	Medicine
2	Digestive system
27	Large intestine
27219	Vermiform appendix

Common subdivisions are marked by lower-case letters, capital letters, or arabic numerals (adding to the confusion). The punctuation marks serve as facet indicators with the following meanings:

(.)	connecting symbol for Personality
(:)	connecting symbol for Matter
(@)	connecting symbol for Energy
(^)	connecting symbol for Space
(~)	connecting symbol for Time

The following examples demonstrate the synthesis process and the complexity of the notation:

1. M7:8.56163'N66 Textile printing in Lancashire in 1966

M	useful arts
M7	textiles
:8	textile printing
.56163	Lancashire
'N66	1966

2. V2,6:2'N5 Constitution of local bodies in India to 1950

V	history
2	India
,6	local body
:2	constitution
'N5	1950

The concepts of faceted analysis and concept synthesis developed by Ranganathan are major contributions to modern classification theory. LaMontagne [1961] compares enumerative classification schemes to tourist conversation books which are composed of ready-made phrases and sentences, while faceted systems are more like dictionaries which a classifier can use to construct subjects as needed. Gopinath [1972] notes that the faceted classification model is better suited to automated applications than enumerated schemes. Organized as separate lists (facets) of simple terms with a prescribed combination order, automation of the faceted model is potentially a simple process. The complex notation and intricate synthesis rules of CC, however, have prevented its widespread use in large libraries.

#### *2.3.4 Software Classification*

Computer Science classification schemes have developed ad-hoc as the computer field has grown. The first general classification scheme for the computer discipline was not published until 1960 [Ralston 1981; Sammet 1982]. It listed 33 subjects arranged in alphabetical order with no subdivisions.

Software classification schemes can be divided into three categories: (1) general computer science schemes such as CR, (2) program libraries (e.g., GAMS and IMSL), and (3) software directories which list application programs. In general, these schemes are enumerative and are characterized as having poor logical arrangements [Prieto-Diaz 1985].

##### *2.3.4.1 General Schemes*

The ACM Computing Reviews (CR) was the first general computing classification scheme. The first edition, published in 1960, contained 33

subjects enumerated alphabetically. The CR was revised in 1964 and did not change again until the third edition was published in 1982 [Sammet 1982; Ralston 1981]. The 1982 CR scheme is based on the AFIP's Taxonomy of Computer Science and Engineering [AFIP 1980] with an organization similar to the DDC. This enumerated hierarchical scheme is limited to only a three level hierarchy with 11 main subject classes [Sammet 1982]. Lower level classes are not assigned a notational code and are kept as lists of terms to facilitate expansion of the scheme.

The main purpose of the CR classification scheme is to organize collections of computer science literature. Although it was not designed for classifying software components, it has been used as the basis for some software libraries such as the Reusable Software Library for Ada packages developed at Intermetrics, Inc. [Burton et al. 1987].

#### *2.3.4.2 Program Libraries*

Computer program libraries, as discussed in section 2.1, are mostly functional collections of small, single function programs or subprograms for solving specific problems. The typical classification is by the type of problem they solve. The IBM Users Group, SHARE, published one of the first software classification schemes in 1963 [SHARE 1963]. The SHARE scheme is a hierarchical enumerative scheme similar to DDC with 22 main classes grouped according to the type of problem solved [Bolstad 1975]. It is heavily biased toward mathematical software with several of the classes devoted to mathematical problems having up to 6 hierarchical levels. The other main classes (not mathematical) contain only 3 levels.

The Guide to Available Mathematical Software (GAMS) classification scheme developed by the National Bureau of Standards was derived from the SHARE scheme [Boisvert et al. 1983, 1985]. It is a tree structured hierarchy with 19 main classes for mathematical and statistical software. The process used for subdividing the main classes is described as being similar to a course syllabus arrangement [Boisvert et al. 1983]. "Within each we tried to place core subjects before more specialized subjects, as one might find subjects arranged in a course syllabus." This technique resulted in the separation of classes which logically should have been collocated. The IMSL library [IMSL 1984] and SPSS [SPSS 1984] are examples of other program library classifications derived from the SHARE scheme.

#### *2.3.4.3 Software Directories*

The tremendous growth in commercial software packages has created a new spin-off industry of software directories (catalogs). Each different directory has its own unique classification scheme usually based on division by application area. The schemes change continuously (since they are enumerative) attempting to reflect the rapid turnover of commercial software.

The most sophisticated scheme was developed by International Computer Programs (ICP). It is hierarchical with a decimal notation similar to DDC. Classes are based on application area with substantial revisions published twice each year. The schedule of one edition usually cannot be used to locate software in another edition. One change from the 1983 edition to the 1985 edition is illustrated below:

1983		1985	
35	Finance	41	Banking and finance
35.1	Banking	41.1	Bank management
35.1.1	Bank management	41.1.2	Operations support
35.1.1.1	Bank planning		
35.1.1.2	Bank operation		

The continuous revision of classification schemes is due to the large growth in commercial software and the enumerative nature of the schemes. For example, the 1980 edition of the International Directory of Software (IDS) listed 3200 products [IDS 1980]. The 1982 edition [IDS 1982] contained 2250 new or different products for a 70 percent change in only two years. These schemes typically list classes only at a very general level. The 1982 edition of IDS lists 199 programs under class 52.0, Inventory Control. The user is left with the task of inspecting each one to make a selection.

Current software classification resembles the early history of library classification. The schemes are enumerative, inflexible, and too general to be of much help to the user. A different scheme is used for each collection of software with no plans for a single universal classification system.

## 2.4 Summary and Conclusion

This section surveyed the areas of software reusability, information retrieval, and classification theory. Several models of software reuse were examined, establishing the impetus for reusing software components and the need for a software classification scheme and component retrieval system. An analysis of the three information retrieval models revealed several



useful techniques, but overall a disappointing recall and precision performance level.

The history of library classification shows an evolution from enumerative schemes toward the use of faceted techniques. Enumerative schemes are too voluminous, difficult to expand, and time consuming to compile for rapidly changing subject areas. Faceted schemes are easier to expand, facilitate automation, and are significantly smaller. Software classification appears to be following an evolutionary pattern similar to that of library classification. Current software classification schemes are mostly enumerative like the early library schemes and suffer from frequent revisions to accommodate changes. To date, software classification has not taken advantage of the faceted approach. It would appear advantageous to do so, especially for large, heterogeneous software collections. The possibility will be examined in Section 3.

### 3. CLASSIFICATION OF SOFTWARE COMPONENTS

Classification is an organizational tool providing a systematic order to a set of concepts or items, and displaying their relationships. Classification provides the capability to answer membership questions (e.g., to say whether or not a given concept belongs to a certain class or, to describe the characteristics by which it does not belong). One of the reasons large libraries of software components have not been successful is the lack of a relevant organization (i.e., no appropriate software classification scheme). Finding the desired component in a large, unorganized collection is analogous to finding a book in a library which has no card catalog and where all books are arranged alphabetically. Unless the exact title of the desired book is known, locating it will be mostly a matter of chance. A library designed for software reuse also must be organized to aid users in locating a particular component.

The design and development of a software component classification methodology and scheme is presented in this section as a three stage process: (1) definition of the design objectives, (2) selection of an appropriate classification model and set of techniques from the domain of classification theory, and (3) development of the classification scheme and related classification methodology. This section describes and examines each of these stages, concluding with a demonstration of the classification methodology on a collection of software components.

### 3.1 Design Objectives

The primary purpose of this classification scheme is to organize a collection of software components to facilitate reuse. Library science provides a mature classification technology from which general objectives and philosophies for the development of classification schemes can be drawn.

#### 3.1.1 General Classification Objectives

The supporting principles for traditional classification theory were originally derived from the areas of logic and philosophy, specifically from the field of logical division (dividing a class into subclasses). The main classification principles derived from logical division are:

1. The characteristic of division must produce at least two classes (e.g., the characteristic of sex when applied to the class of "humans" does produce two classes; when applied to the class "mothers", it does not produce two classes).
2. Only one principle of division should be used at a time to produce classes. Otherwise, some elements would appear to belong in multiple classes (a cross-classification problem).
3. Subclasses must completely exhaust their superordinate class or all concepts (subjects) will not be classified.
4. No step of division should be omitted when successively dividing a class into smaller and smaller subdivisions.

Langridge [1973] cautions that these principles are an ideal and cannot always be totally satisfied in actual applications. It may be impossible to establish mutually exclusive classes for all items in a collection (principle 2) or even know what the correct number of steps is for dividing a class into subdivisions (principle 4). There is also no absolute judgment about a

particular classification scheme being right or wrong, only that some are better for a given purpose than others [Buchanan 1979; Langridge 1973].

Traditional classification schemes can vary widely in form and organization as described in Section 2. However, in addition to the principles of logical division, most classification schemes recognize the following general objectives [Chan 1981; Buchanan 1979; Vickery 1960]:

1. A classification scheme's main purpose is to help the user identify and locate an item in the collection. This implies a collection may need to be organized in different ways (viewpoints) for different groups of users. The organization should satisfy the needs of the users.
2. A classification scheme should group like subjects together based on certain characteristics. The characteristics may vary according to the purpose of the scheme, but related classes should be collocated in each scheme.
3. The organization should be hierarchical where possible. Classes which depend on other classes, are developed from other classes, or are developed later than other classes should generally follow them in the hierarchy.
4. The schedule (list of all classes and their relationships in a prescribed order) should contain terms which are generally used and understood by the intended user community.
5. The scheme should be flexible since changes and expansion will always occur. Rigid systems can dislocate their general sequence of classification when new terms are "patched" into full classes, and eventually entire categories must be re-classified (given a new mapping) when the order becomes too irregular.
6. The citation order (order of combining classes) must support the first three objectives. Classes cited first will be collocated while those cited last will be scattered within the collection. Increasing quantity, dependence order, part to whole, concrete before abstract, later in time, later in evolution, spatial, and alphabetical are examples of possible citation orders. Since citation order determines the arrangement of subjects within the collection, it must be based on the needs of the users.

7. The scheme should facilitate automation. Many of the traditional systems are based on illogical organizations or use such complex notations that automation is practically impossible.

8. A concise and simple notation (set of symbols used for identification of classes, the shelving of documents, and arranging entries in a catalog) is an important objective of library schemes since they are concerned with the physical location of documents and books. Complex notations have caused the obsolescence of several schemes. However, since physical collocation of software components is not important, a separate notation is not needed for software classification.

### *3.1.2 Software Classification Objectives*

The objectives and principles outlined above provide a general set of guidelines for the design and development of classification schemes. As they were specialized to support the specific arena of software component classification within a reuse environment, some of the objectives were modified based on the following assumptions: (1) large collections encompassing thousands of software components will exist, (2) reuse will become a common practice, and (3) reuse will expand beyond the code level to include all products from the software development life cycle (i.e., specifications, designs, requirements). The specialized objectives are presented below.

1. The software component classification scheme's main purpose is to assist the user by identifying and locating a particular set of potentially reusable components.
2. The structure and concepts used in the scheme should be similar to those used by software developers working in a particular application area.
3. The scheme and descriptive attributes developed should provide precision for discrimination among similar components.

4. The scheme should support an automated retrieval mechanism which provides relevance ranking of retrieved components.
5. The citation order should be adjustable to meet the needs of different users. This implies a reordering of classes to adapt to changes in importance. For example, one user may consider language an important attribute while another needs performance as the main criteria for selection.
6. Flexibility and expandability will be important considerations. Computer science is a rapidly changing field. The scheme should be easy to modify (expand) without causing re-classification problems.

These objectives and principles were used to compare and judge the basic classification models. The selected model then served as the foundation for the development of a software component classification scheme and methodology.

### **3.2 Selection of a Classification Model**

There are only two basic models for classification schemes. The first type, enumerative, lists every subject in the schedule that can possibly occur. It is the more traditional method characterized by the division of knowledge into successively narrower classes including all the elemental, superimposed, and compound subjects which the scheme can classify. The classes are usually arranged in an order displaying their hierarchical relationships. The Decimal Classification is an example of an enumerative scheme. It divides the universe of knowledge into ten main classes, each main class into ten subclasses, each subclass into ten, and so on.

The second method, faceted, is a product of modern classification theory. It uses a building block approach which composes subjects from

their elemental terms. Faceted schemes list only basic (elemental) subjects arranged in categories (facets) along with their generic relationships. The elementary terms can then be combined to form compound subjects (synthesis). Faceted schemes are also called "analytico-synthetic" to describe the techniques involved in their development and use.

Both of these classification models will be evaluated using the general and specific objectives listed in the previous section with particular emphasis placed on the software component classification objectives.

### *3.2.1 Identification, Location, and Precision*

The first objective (general objective 1 and specific objective 1) states a classification scheme's main purpose is to assist the user in identifying and locating an item (or set of reusable components) in the collection. The synthesis capability in the faceted scheme is well suited to this purpose. Highly detailed and complex subjects can be described with the exact amount of precision required (specific objective 3).

Enumerative schemes cannot provide the same level of detail for large collections. The fixed citation order in enumerative schemes can even hinder location of a desired item. Buchanan [1979] provides an example of this problem from DC18, a widely used enumerative scheme. Figure 10 shows a portion of the Music class from the DC18 schedule. The citation order hinders location of certain subjects by scattering them illogically. For example, works on brass instruments in general are separated from works on particular brass instruments, such as trumpets, by the class of woodwind instruments. Similarly, works on single-reed instruments are separated from

788	Wind instruments
788.01	Brass instruments
788.05	Woodwind instruments
788.056	Reed instruments
788.1	Trumpets
788.2	Trombones
788.4	Horns
788.5	Flutes
788.6	Single-reed instruments
788.62	Clarinets
788.7	Oboes
788.8	Bassoons

**Figure 10.** DC18 Music Class

those on reed instruments in general by classes about individual brass instruments. The class flutes, a woodwind instrument, is confusingly located in the middle of the brass instruments. It is illogical that the class SINGLE-REED INSTRUMENTS (e.g., clarinets, saxophones) has been enumerated, but the class DOUBLE-REED INSTRUMENTS (e.g., oboes, bassoons) is not listed although many of the individual double-reed instruments are enumerated. This example demonstrates how an enumerative scheme can hamper the identification and location of some items. A collection of software components using this type of organization would not promote reuse.



### *3.2.2 Collocation, Organization, and Structure*

Both enumerative and faceted schemes can support a hierarchical organization equally well, where appropriate (general objective 3). However, faceted schemes are more adaptive to the structure and concepts familiar to the users (general objective 4 and specific objective 2) since only elemental classes and their generic relationships are listed in the schedule. Enumerative schemes must list each possible compound class ready-made, a time consuming and complicated process frequently causing collocation problems and omission errors (general objective 5).

The citation order of a faceted scheme can easily be changed to meet the needs of different users (specific objective 5) since only generic relationships between facets (representing a very simple hierarchy) are involved. The fixed citation order in enumerative schemes cannot be changed without a complete (and massive) re-classification which would be prohibitive for large collections.

### *3.2.3 Expansion and Automation*

Computer software is a rapidly changing discipline. This requires the classification system to be easy to modify and expand (general objective 6 and specific objective 6) without associated re-classification problems. If the elemental classes for a new compound class are already present in a faceted scheme, inserting the new class is trivial (nothing has to be done at all). If the new compound class contains new elemental classes, they only have to be added to the correct facet or subfacet. No re-classification is involved. In an enumerative scheme, new compound or superimposed classes must be given

special locations. When new elemental classes are added, all the new superimposed and compound classes they will generate must also be added. This often causes the hierarchy to become malformed as it is forced to grow deeper in certain disciplines when new classes cannot be added at the appropriate level.

A classification scheme should facilitate automation in general (general objective 7) and, for the software component reuse arena, an automated retrieval mechanism with relevance measuring capabilities (specific objective 4). Enumerative schemes in general are not easy to automate. Certain sections of enumerative schedules usually contain illogical organizations (collocation problems) or malformed hierarchies resulting from expansion difficulties. These problems destroy the general principles and structure which make automation possible. When these problems are combined with the complex classification rules and methods needed for most enumerative schemes, it is easy to see why automation is not usually attempted.

Faceted schemes provide a model which is better suited to automation. Organized as separate lists (facets) of terms with well defined combination orders for synthesis of superimposed and compound classes, automation is potentially a simple process. The generic hierarchy between facets provides a ready-made tool for determining relative relevance measures. For example, it would be possible to program a search for the concept "England" in a UDC library (faceted) since it is always given the notation "(42)". However, the same search in DC (enumerated) would not be

possible. In DC, England can be specified using several different notations which also have other meanings (e.g., are overloaded).

### *3.2.4 Selection Summary*

An analysis of the design objectives reveals the enumerative model satisfies only a handful of the requirements. Enumerative schemes can be intractable or even hinder location of a desired item, frequently fail to collocate related items, are not easily changed (flexible) to meet the needs of different users, are difficult to expand without re-classification problems, and do not facilitate automation. The survey of existing schemes for software classification in Section 2.3 confirms the disadvantages of this model for software component classification.

The faceted model, on the other hand, satisfies all of the general and specific design objectives except one, the requirement for a simple notation (general objective 8). The synthesis process provides a detailed descriptive capability for complex subjects, but this often results in a lengthy, complex notation. However, since notation is not necessary for software components (i.e., physical collocation is not important), this is not a serious disadvantage. A faceted scheme can accommodate an expanding collection of software components providing precise classification of highly specific subjects, flexibility to meet user needs, and a good basis for automation.

## **3.3 A Faceted Scheme and Classification Methodology**

This section presents a faceted methodology and classification scheme developed as part of this research for classifying software components. First,

the basic concepts to be used during the classification process are explained. Then the methodology is described with examples demonstrating each stage. A complete classification scheme is presented in the final section.

### *3.3.1 Basic Concepts*

A faceted classification scheme lists only elemental classes, differing from the more traditional schemes which assign fixed slots to all subjects (elemental or compound) in an enumerative sequence. The elemental classes, called foci, are organized into categories called facets. A facet contains all the foci which are kinds of the same things. It is possible to have groupings of foci within a facet called subfacets. Subfacets are a product of the single characteristic principle of logical division. Subfacets are created by the application of one characteristic of division on the foci within a facet [Buchanan 1979]. Combination order is the order facets are cited to form a composite subject, and citation order is the overall order of the classification schedule (the order of facets, the order of subfacets within facets, and the order of foci within subfacets).

### *3.3.2 The Methodology*

A faceted classification methodology involves two main processes as the name "analytico-synthetic" implies. The first part, analysis, is the construction of the faceted scheme using the technique of literary warrant. Figure 11 illustrates this schedule construction process. Using literary warrant, subjects to be classified are analyzed into their elemental terms, and these terms are then grouped into homogeneous, mutually exclusive facets. Each facet is determined by a single characteristic of division. Facet

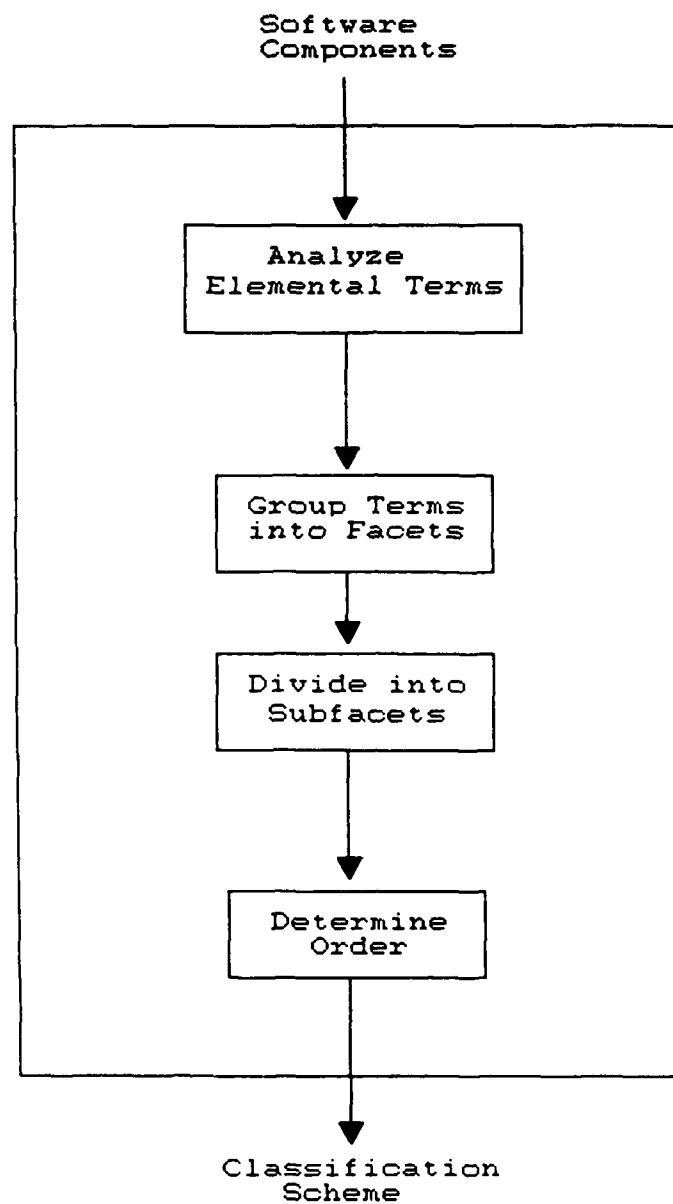


Figure 11. Constructing a Faceted Classification

analysis is therefore analogous to the traditional rules of logical division. After the facets and subfacets have been defined, the citation order is determined using the objectives and principles described in section 3.1.

General guidelines for the construction of faceted schemes based on literary warrant abound in library science literature [Buchanan 1979; Vickery 1960; Prieto-Diaz 1985; Chan 1981]. These guidelines, however, are usually designed for document classification systems. The methodology described below is a consolidation of these techniques, specialized for application to software component classification.

1. Analyze a representative sample of the software components to be classified into their elemental classes.
2. Group the elemental classes into facets using the single characteristic principle of logical division.
3. When necessary, apply different characteristics of division within facets to produce subfacets. This may become an iterative process and include the regrouping of facets.
4. Select an appropriate order for foci within the facets and subfacets based on the needs of the users.
5. Put the subfacets in order within their facets.
6. Determine the combination order for the facets which will be used in constructing compound subjects (a facet formula).

The second part of a faceted scheme is the synthesis of superimposed, complex, or compound subjects. This involves assembling the elemental classes in the prescribed order to express the desired subject. Both parts of this methodology will be explained in more detail in the following sections as the methodology is used to construct a classification scheme.

### 3.3.3 *The Faceted Scheme*

In this section, the methodology developed in the previous section is described in detail as it might be applied to a specific set of reusable software components. The first step in the classification process, analysis of terms, is actually preceded by a preparatory step, the selection of a subject field. To provide a manageable context for this phase of the research, the familiar area of canonical data structure algorithms (i.e., searching, sorting) was chosen. Sedgewick [1983], Tenebaum and Augenstein [1981], Wirth [1976], and Knuth [1973] were the primary sources for this collection of components. The following sample set containing five algorithms will be used to demonstrate the faceted methodology.

1. A selection sort written in Pascal which sorts an array of integers. The routine was written for the VAX AOS environment and sorts over 2000 values per second.
2. A routine to sort strings of characters contained in an array. It is written in Fortran for the IBM VS environment and uses a simple bubblesort technique. It requires over 500K bytes of memory.
3. A specification for a search algorithm written in Ada. The algorithm searches for a given word in a binary tree using a preorder search technique. Each node in the tree contains a single word.
4. A Basic quicksort routine that runs on an IBM/PC compatible MS/DOS microcomputer sorting about 2500 real numbers per second. The numbers are held in a linked list.
5. An Ada routine to sequentially search a linked list and locate a particular word. Each node contains one word but there is no limit on the number of nodes allowed. It was developed on a VAX using AOS.

### 3.3.3.1 The Analysis Phase

The construction process begins with an analysis of a representative sample from the target collection to identify the elemental classes. During this stage, it is important to recognize synonyms and homonyms. Only one class should be created for each concept (synonyms), but it is also necessary to ensure that different concepts form different classes (homonyms). An analysis of algorithm (1) provides the foci PASCAL, SORT, ARRAY, INTEGER, SELECTION, VAX, AOS, and 2000 PER SECOND. Algorithm (4) yields the additional foci BASIC, QUICKSORT, IBM/PC, MS/DOS, SORT, REAL NUMBERS, 2500 PER SECOND, and LINKED LIST. The complete analysis of all five routines produced the following list of unordered elemental classes:

Pascal	Basic	bubblesort
sort	quicksort	500K bytes
array	IBM/PC	Ada
integer	MS/DOS	sequential
selection	real numbers	word
Vax	linked list	node
code	strings	binary
AOS	Fortran	tree
2000/second	specification	preorder
2500/second	VS	search
character		

Consolidation or elimination of some terms may occur during subsequent steps in the process, but it is important to identify every elemental class possible during the analysis stage.

Step 2 is the grouping of foci into facets. This action is based on the single characteristic of division principle combined with general knowledge about the subject area and needs of the users. From the list of simple classes,



AD-A196 541

A CLASSIFICATION METHODOLOGY AND RETRIEVAL MODEL TO  
SUPPORT SOFTWARE REUSE(U) AIR FORCE INST OF TECH  
WRIGHT-PATTERSON AFB OH D L RUBLE 1988

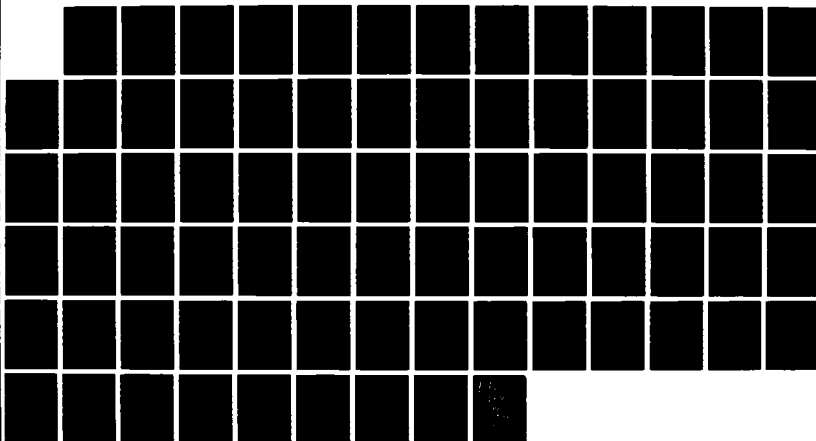
2/2

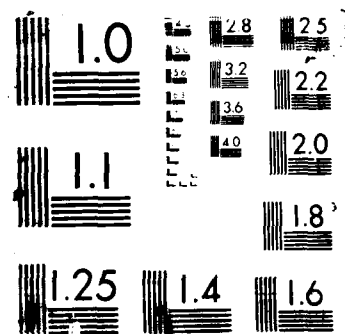
UNCLASSIFIED

AFIT/CI/MR-88-58

F/G 12/5

NL





several facets such as language, algorithm, and data structure are suggested.

Facet groupings for the example collection are given below:

(Language facet)

Pascal  
Basic  
Fortran  
Ada

(Data Structure facet)

array  
linked list  
node  
tree

(Algorithm facet)

selection  
quicksort  
bubblesort  
sequential  
binary  
preorder

(Operating System facet)

AOS  
MS/DOS  
VS

(Hardware facet)

Vax  
IBM/PC

(performance Rate facet)

2000/second  
2500/second

(Memory Requirement facet)

500K bytes

(Activity facet - the function of the component)

sort  
search

(Focus facet - object that is the focus of the activity)

integer  
real  
string  
word  
character

(Form facet - the type of the component)  
code  
specification

This first approximation of facets is based on a small collection of information. The characteristics of division may be changed during subsequent stages of the development process as more items are added to the collection and more information becomes available about the needs of the users.

The next stage determines if any of the facets can be divided by additional characteristics to form subfacets. The logical division principle concerning the application of only one characteristic of division at a time is used during this step. Ignoring this principle can create cross-classification problems, or fail to provide a class for a concept as the following sequence from the London Education Classification demonstrates [Buchanan 1979].

.  
.  
.  
Teenagers  
.  
.  
.  
(educands beyond usual age of formal education)  
  
Adult  
Parent  
Housewife  
Older person

These foci are from the Educands facet (people who are being educated). It is obvious these classes are not mutually exclusive. A housewife may be an older person, a parent, and an adult. More than one characteristic was used to produce this set of foci. Three of the foci are classes determined by age, one focus represents occupation, and the other people by relationship.

There is no indication of the preferred class for superimposed or compound subjects (cross-classification) and it is not possible to express the class TEENAGERS WHO ARE PARENTS (missing class).

The formation of subfacets should also take into account the purpose of the schedule. For locating reusable components, it appears that only two facets in this small demonstration collection need to be divided into subfacets: Performance Rate and Memory Requirement. Performance Rate contains classes defined by amount (2000 and 2500) and by time unit (second). Similarly, the possible characteristics of division in the Memory Requirement facet are memory amount (500K) and type of measurement (bytes). Adding a few additional elemental classes to make the example more meaningful and dividing these two facets into subfacets produces:

(Performance Rate facet)

*Performance amount (subfacet)*

1-99  
100-999  
1000-1999  
2000-2999  
3000-3999  
4000-4999  
5000 or more

*Performance time unit (subfacet)*

per minute  
per second  
per microsecond

(Memory Requirement facet)

*Memory amount (subfacet) - expressed in K*

1-99  
100-199  
200-299  
300-399  
400-499  
500-599

600-699  
700-799  
800-899  
900 or more

*Type of memory measurement (subfacet)*

words  
bytes

This completes the process of subfacet grouping in this sample collection. In larger collections, the division process can be iterative with subfacets being divided into subfacets until all possible characteristics of division have been exhausted.

The next two steps in the process place the foci and subfacets in order within facets. This will be demonstrated in Section 3.4 with a larger collection of components. The final step then, is the choice of combination order between facets. Combination order determines the order in which elemental classes are cited when constructing superimposed, compound, or complex classes. While there are general principles for citation order suggested in the literature (i.e., decreasing concreteness, principle of purpose, consensus, wall-picture), the need of the users is always of primary concern. The general principles of order are only used when the needs of the users cannot be determined or vary too widely to be accommodated by a single order. The purpose of the scheme developed in this research is to aid users in locating a software component for reuse. Therefore, FORM is the most significant facet in the collection, corresponding to the phase of the development cycle the user is interested in. Purpose and consensus suggest the ACTIVITY (function) of the component as the second facet. A large number of texts and software catalogs use this arrangement. It then seems

logical to follow the function with the main object of the action and then with that object's location. This places the FOCUS facet third, followed by the LOCATION facet.

The remaining facets are not as easy to order. Their use and importance is not based on any general principle, and user needs in this area vary to a large degree. An adjustable citation order is needed for these facets, and this will be an important feature of the automated model. At this point, an order based on the author's criteria will be used to complete the schedule as listed below:

(Form facet)	(Hardware facet)
code	Vax
specification	IBM/PC
(Activity facet)	(Performance rate facet)
search	<i>Performance amount (subfacet)</i>
sort	1-99
(Focus facet)	100-999
integer	1000-1999
real	2000-2999
string	3000-3999P
word	4000-4999
character	5000 or more
(Location facet)	<i>Performance time unit (subfacet)</i>
array	per minute
linked list	per second
node	per microsecond
tree	
(Language facet)	(Memory Requirement facet)
Ada	<i>Memory amount (subfacet)</i>
Basic	(expressed in K)
Fortran	1-99
Pascal	100-199
	200-299
(Algorithm facet)	300-399
binary	400-499
bubblesort	500-599
preorder	600-699

quicksort	700-799
selection	800-899
sequential	900 or more
(Operating system facet)	<i>Type of measurement (subfacet)</i>
AOS	words
MS/DOS	bytes
VS	

### 3.3.3.2 The Synthesis Phase

A faceted classification schedule contains only elemental classes. Synthesis is used to assemble superimposed, complex, and compound classes. The synthesis process is based on the combination order for the scheme. This allows a subject to always be consistently classified. The following component will be used to illustrate the synthesis process.

This Pascal routine sorts an array of integers using the selection sort technique. It was written for the IBM/PC environment using MS/DOS.

Using the demonstration schedule developed above, the first facet in the combination order is FORM. The elemental class from FORM which describes this component is "code". The second category in the facet formula is ACTIVITY which provides the term "sort". Next the FOCUS class "integer" would be added to the descriptor followed by the LOCATION term "array". The synthesized description assembled thus far is:

code (Form), sort (Activity), integer (Focus), array (Location)

Continuing with the synthesis process, the LANGUAGE facet supplies the term "Pascal" and the ALGORITHM class is "selection". From OPERATING SYSTEM, the elemental term "MS/DOS" would be selected



and "IBM/PC" describes the **HARDWARE** category. This software component does not require classes from the **PERFORMANCE** facet or the **MEMORY** facet. Thus, a complete class description for this software component is given by the terms:

code (Form), sort (Activity), integer (Focus), array (Location),  
Pascal (Language), selection (Algorithm), MS/DOS (Operating system),  
IBM/PC (Hardware)

### **3.4 Application of the Faceted Methodology**

The preceding section described the faceted methodology for constructing a software classification scheme, and applied it to a demonstration library of five components for illustration. This methodology has also been used in the domain of data structure algorithms with the objective of producing a reasonably complete schedule which can be easily expanded. As noted by Vickery [1960], a schedule is never complete; "the task is never completed - new discoveries and hence new terms will occur unceasingly - so that the classifier can never produce a complete schedule."

The procedure used for the analysis stage of this scheme included examination of data structure and algorithm texts to identify as many elemental terms as possible. This list of terms was then augmented by descriptors from software directories and other software classification schemes resulting in a list of 325 terms. Eliminating synonyms reduced this list to 277 elemental classes. During step 2, the grouping of terms into facets, a new facet (**PRECISION**) was identified to characterize the numerical precision of certain components. Subfacet division, the third step, produced several additional subfacets. The **ACTIVITY** facet was divided into "Simple

Function" and "Application" subfacets, FOCUS produced "Simple Object" and "Area" subfacets, and "Data Structure" and "Device" subfacets were identified in the LOCATION facet.

The discussion of foci order within facets was deferred in Section 3.3.3 because the example was too small to demonstrate it properly. This larger collection of 277 classes provides a better context for viewing foci arrangement with the following conclusions. First, none of the traditional principles of order such as chronological order, developmental order, spatial order, order of increasing complexity, or order of increasing size provides a satisfactory order over all the foci in the schedule. Closer examination of the terms reveals small clusters with various relations, but no overall principle of ordering for the entire scheme to indicate that one term should precede or follow another. Vickery [1960] suggests an alphabetical order be used in such situations, although it is not generally favored over the other ordering principles. In this type of situation where no other ordering applies, alphabetical order does at least provide a recognizable order for the user and an arrangement to guide later additions to the scheme.

Step 5, the ordering of subfacets within facets, was also deferred in Section 3.3.3. The principle of increasing complexity is a logical choice for subfacet order in this collection, placing simple function before complex applications in the ACTIVITY facet and simple objects before larger areas of concern in the FOCUS facet. The suggested schedule for data structure algorithms including all 277 foci covers several pages and is included as Appendix 1. The order of facets and subfacets is summarized below in outline form.

**Form**  
**Activity**  
    *simple function*  
    *application*  
**Focus**  
    *simple object*  
    *area*  
**Location**  
    *data structure*  
    *device*  
**Language**  
**Algorithm**  
**Operating system**  
**Hardware**  
**Performance rate**  
    *performance amount*  
    *performance time unit*  
**Memory requirement**  
    *memory amount*  
    *type of memory measurement*  
**Precision**

The construction of this schedule demonstrates the utility of the faceted methodology as an organizational tool for collections of reusable software components. The general design principles of collocation, description and identification, use of familiar terms, and a citation order based on user needs were verified. The remaining objectives concerning retrieval automation, an adjustable citation order, and schedule expansion will be examined in Sections 4 and 5 using this schedule for data structure algorithms as a baseline.

#### **4. SOFTWARE COMPONENT RETRIEVAL**

Software component reuse offers significant benefits and savings, but is only practical if developers can easily locate a component to reuse. Locating a particular component in a large software collection requires the application of three cooperating processes: (1) an organization for the collection, (2) a method for describing components based on that organization, and (3) a mechanism to access (locate and retrieve) the desired component or set of components. The faceted classification system presented in Section 3 provides two of the necessary elements: the organization and descriptive capability. This section presents the design and implementation of the third element, the retrieval mechanism. The functional requirements for a software component retrieval mechanism are specified in Section 4.1. Sections 4.2 and 4.3 discuss the design of the formal retrieval model and the design of the automated retrieval system, respectively. The design incorporates the classification scheme for data structure algorithms which was presented in Section 3. The implementation of this design as a rapid prototype is presented in the final section.

##### **4.1 Functional Requirements**

The main objective of this software component retrieval system is to return a set of reusable components which satisfy a given description of the characteristics of a desired component. The functions required to provide this capability include query formulation, relevance ranking of retrieved components, a method for vocabulary control, and provisions for user reordering of the software collection.

#### *4.1.1 Query Formulation*

The retrieval mechanism should provide adequate support for query formulation. Queries should be concise and easy to form, yet describe the characteristics of the desired component as completely as possible to distinguish it from similar components. These two objectives have conflicting realizations. As the size of the descriptor in the query increases, so does its descriptive power (resolution). When the descriptor is small, the precision decreases. This relationship suggests a compromise is necessary. Both objectives cannot be maximized at the same time. The descriptor size should be short, yet fully describe the relevant attributes of the component. Increasing the descriptor size to increase precision can produce long, complex descriptions. For example, some specification languages used to describe programs produce descriptions as large or larger than the original programs they seek to describe [Balzer 1985; Horowitz and Munson 1984; Nourani and Jones 1985]. Such lengthy descriptors may reduce the benefits of reusability.

Bartschi [1985] suggests a similarity relation exists between concepts expressed in the query and concepts in the items of the collection which imposes a limit on the possible query forms. Query forms should correspond to the representation of items in the collection. Natural language, sets of query descriptors, and descriptors combined by operators are the typical forms of queries in most retrieval systems. Whatever the representation, query formulation must answer the questions: How can the desired item be described? How should the underlying information structures be represented? How does the user find sensible query terms to use? What

means are provided to interconnect query terms? Which combinations of query terms can retrieve relevant items and reject irrelevant ones? An effective software component retrieval system should address each of these requirements.

#### *4.1.2 Relevance Ranking*

The retrieval mechanism should not simply locate and return only components which exactly match the query. Selected items should be ordered based on their relevance to the query. Relevance ranking can assist the user in selecting the best component to reuse. Return of an unordered group of components places the evaluation burden on the user, making selection of the appropriate item a tedious manual process and negating the benefits of reuse (or eliminating reuse altogether). The "all-or-nothing" approach of traditional boolean systems is an example of unordered retrieval. The retrieved items in boolean systems are indistinguishable from one another and are all treated as equally relevant to the query.

Information retrieval research has produced a variety of techniques for relevance estimation including term weighting, threshold values, fuzzy set theory, clustering techniques, and similarity measures. Term weighting is used to express the relative importance of a concept. It can be applied to either the index term (descriptor) of an item or to a query term. Query term weighting is used to indicate the relative importance of a descriptor in a query. For example, the query ' $t_1$  AND  $t_2$ ' gives no indication of which descriptor,  $t_1$  or  $t_2$ , is more important to the user. The weighted query " $(t_1, 0.8)$  AND  $(t_2, 0.3)$ " however, supplies the additional information that items with characteristic  $t_1$  are more relevant to the user's needs than items having

only characteristic  $t_2$ . Similarly, items possessing both concepts,  $t_1$  and  $t_2$ , are considered more relevant than items having only characteristic  $t_1$  or only characteristic  $t_2$ . Query term weighting can also be used as a threshold value. Items in the collection are judged based on whether or not they reach a certain threshold for each characteristic and by what degree they surpass that level. Threshold values are accumulated for all query descriptors and then are used to order the items which exceed the threshold value.

Fuzzy set theory has been applied to retrieval systems in a variety of ways as a means of providing relevance measures. The objective is to estimate the degree of fit between the query and each item in the collection as a fuzzy set with unsharp boundaries. The "all-or-nothing" retrieval paradigm is replaced by a process with intermediate degrees of matching. Several combinations of weighting schemes and measurement formulas have been suggested in the IR literature for use as fuzzy set mechanisms. Clustering techniques are usually added to these systems to reduce the search space and computational workload when searching large collections. Items in the collection with similar characteristics are grouped together in a cluster. Clusters are then treated as single entities during the initial search process. Clustering is similar to the collocation principle used in classification theory.

#### *4.1.3 Vocabulary Control*

The retrieval mechanism should provide vocabulary control and a well-structured index language. When using a retrieval system, a user has a better chance of locating the desired item if the terms in the query are the same as those used by the system to describe that item (e.g., an item indexed

by "ship" will not be retrieved by a query containing the term "boat"). The terms which can be used to describe items in a collection form an index language ( a language used to describe items and queries). In IR technology, two types of index languages are recognized: pre-coordinate and post-coordinate [Van Rijsbergen 1979]. Pre-coordinate indicates that terms are coordinated at the time of index creation, while post-coordinate means that terms are coordinated at query formulation time. The vocabulary of an index language may also be controlled or uncontrolled. A controlled vocabulary is a list of approved terms to be used with a particular collection. Other descriptors are not allowed as index terms in a controlled vocabulary. The general guidelines available concerning index languages indicate that simple languages without overly complex controls are preferred. The author's experience with large commercial retrieval systems suggests that vocabulary control can significantly improve retrieval performance.

Developing an index language requires decisions in several areas. A method must be selected to distinguish between homographs (terms with the same spelling which may or may not be pronounced the same). One common technique is to qualify the terms with another word (e.g., PITCH(bitumen), PITCH(music), PITCH(baseball), PITCH(slope)). Failure to distinguish homographs can result in reduced retrieval precision. Semantic relationships such as equivalence (synonyms) and hierarchy (genus to species, or part to whole) should also be included in the index. One frequently used technique for handling semantic relationships is a systematic arrangement which groups related terms together (i.e., a classification scheme) [Foskett 1982]. Well-formed semantic relationships in an index language can improve query precision and recall performance.



#### *4.1.4 Collection Reordering*

An automated retrieval system should be able to reflect user specified query term importance. One way to provide this capability is to reorder the categories in the collection to indicate the relative importance of each group. The semantic arrangement (collection order) of many retrieval systems is static, reflecting only the criteria of the developers. Users with differing needs do not always agree with these criteria. If the order of the collection is fixed, it will not serve the needs of these users effectively. The capability to reorder categories to reflect the importance of each group to an individual user would improve both precision and recall. An automated retrieval system should be able to provide this capability via an adjustable citation order.

### **4.2 Design Decisions for the Retrieval Mechanism**

A retrieval mechanism to support software component reuse should satisfy the functional requirements outlined above. The design decisions for the retrieval mechanism included: selecting a retrieval model, query formulation and internal representation, selecting a similarity measure for relevance ranking, and collection reordering to reflect individual user's needs. These design decisions and accompanying rationale are discussed below.

#### *4.2.1 The Retrieval Model*

Selection of a retrieval model to use as the basis for the retrieval mechanism was a major design decision. The three models (boolean, vector

space, and probabilistic) described in Section 2.2 were analyzed using the functional requirements outlined in Section 4.1. The functional requirements combined with the probabilistic model's incomplete definition quickly narrowed the possibilities to either the boolean or vector space model.

Boolean technology offers simple and flexible query formulation, and the internal processing structure based on inverted file manipulation is extremely efficient. These features make the processing of large collections possible. However, the boolean model does not support importance indicators for query terms or the use of similarity measures for relevance ranking. The basic boolean model alone does not satisfy the functional requirements for software component retrieval.

The vector space retrieval model was developed in response to the disadvantages of boolean retrieval. Weighting of index and query terms to indicate importance, and relevance ranking of selected items are directly supported. But to provide these capabilities, the vector space model loses the simple boolean query structure, and the selection of an appropriate similarity measure to provide relevance ranking is a problem as noted in Section 2.2.1.3. The large amount of additional computations required for query evaluation limits vector space retrieval systems to small collections.

Neither model supports all the requirements for a software component retrieval mechanism. Therefore, a hybrid model was designed in this research which combines the basic boolean and vector space retrieval models, incorporating the organizational structure provided by a faceted classification scheme. Using the vector space approach, each software

component in this hybrid retrieval model is represented as a tuple of attributes

$$C_k = \langle c_1, c_2, \dots, c_i, \dots, c_n \rangle$$

where each  $c_i$  is an attribute term from some facet  $F_i$ . Each query is similarly defined as a tuple of attributes:

$$Q_t = \langle a_1, a_2, \dots, a_i, \dots, a_n \rangle$$

This representation for components and queries provides potential capabilities for term importance weighting and component relevance ranking. The design uses an ordering on the facets (either the citation order or user supplied order) to indicate term importance and to measure relevance as the following example illustrates. Let the ordering  $F_1, F_2, \dots, F_i, \dots, F_n$  indicate facet  $F_i$  is more important than facet  $F_j$  whenever  $i < j$ . A query expressed as  $q = \langle a_1, a_2, a_3 \rangle$  would indicate term  $a_1$  (from Facet  $F_1$ ) as most important and term  $a_3$  (from Facet  $F_3$ ) as least important.

An efficient matching function is one of the advantages of the boolean model. Modifying that basic model to reflect term importance by ordering the attributes yields a matching function  $M$  represented as:

$$M(q_t) = \{C \mid c_1 = a_1 \wedge c_2 = a_2 \wedge \dots \wedge c_n = a_n\}$$

A match is indicated when attribute  $c_1$  of a particular software component tuple matches query attribute  $a_1$ , and component attribute  $c_2$  matches query attribute  $a_2$ , and so on for all the attributes in the tuple. However, this function will return the empty set if there are no exact matches. Retrieving components which almost match the one requested and ordering them by

relevance requires the use of a similarity measure as in the vector space model.

#### *4.2.2 Query Formulation and Vocabulary Control*

The two major requirements for query formulation are: (1) queries should be concise and easy to form, and (2) queries should adequately describe the desired component. In the retrieval model designed in this research, the index language uses elemental terms from each facet in the software classification schedule to form the query tuple. This design satisfies both objectives. The elemental terms from the classification schedule provide users with ready-made descriptors. For example, in the query:

$$q = \langle \text{code, search, word, tree, Ada, VS} \rangle$$

the term "code" was obtained from the FORM facet, "search" from the ACTIVITY facet, and so on. Since the software components in the collection were classified using the faceted scheme, queries formed from the same schedule should be able to fully describe the desired components.

This index language formed by using terms from the classification schedule provides additional features. It is pre-coordinated (coordinated at the time of indexing) and controlled (limited to the terms in the schedule). Homographs, synonyms, and term hierarchies are identified and included in the schedule when it is constructed.

### 4.2.3 A Similarity Heuristic

The matching function,  $M$ , defined in Section 4.2.1 can only determine exact matches between the set of software components  $C$  and the set of user queries  $Q$ . A similarity measure is needed to locate components which almost match the query but are not exact matches. Vector space retrieval supports this closeness principle, but the similarity measures suggested in the IR literature have not been validated. The geometry-based measures used in vector space research can sometimes produce unpredictable and erratic results. A heuristic approach was used to design a more appropriate similarity measure for software components.

Closeness, when selecting a software component for reuse, implies a match on critical attributes in the query, but possibly not on every characteristic. A component not matching "all" of the desired attributes may still be reusable, if the nonmatching attribute(s) is not critical or can be compensated for easily. For example, the query

$$q = \langle \text{code, search, word, tree, Ada, VS} \rangle$$

indicates the attributes "code" and "tree" are more important than the operating system attribute, "VS". Given the components

$$C_1 = \langle \text{code, search, word, tree, Ada, AOS} \rangle$$

$$C_2 = \langle \text{code, insert, word, tree, Ada, VS} \rangle$$

it is easy to determine that  $C_1$  is more relevant to the reuser's query  $q$  than component  $C_2$ . All attributes in  $C_1$  match the query exactly except the least important, the operating system. Reusing component  $C_1$  in this situation

would not require much additional effort.  $C_2$  also matches all attributes in the query except one. However, the unmatched characteristic is the second most important attribute, that is the function of the code, making reuse unacceptable.

The similarity heuristic described above was used to extend the boolean matching function defined in the previous section. A "match-anything" characteristic, shown as "\*" in the following examples, was added. The extended matching function with the "match-anything" characteristic shown for facet<sub>i</sub> becomes

$$M(q_t) = \{C \mid c_1 = a_1 \wedge c_2 = a_2 \wedge \dots \wedge (c_i = a_i \vee c_i = *) \wedge \dots \wedge c_n = a_n\}$$

and can be used in conjunction with facet ordering to measure component relevance to a given query. In this design, the retrieval mechanism automatically invokes the similarity heuristic which alters the internal representation of the query by inserting the match-anything characteristic successively, starting with the least important term. This allows the retrieval mechanism to search for items which almost match the query. To illustrate, when the similarity heuristic is applied to a query  $q = \langle a_1, a_2, a_3, a_4, a_5 \rangle$ , after exact matches have been found  $a_5$  would be replaced by \* and the search repeated, then  $a_4$  and so on. Thus for the two components

$$C_1 = \langle a_1, a_2, a_3, a_4, * \rangle$$

$$C_2 = \langle a_1, a_2, a_3, *, a_5 \rangle$$

both matching a given query except for the "\*" characteristics, then component  $C_1$  will be judged more relevant to the query than component  $C_2$ .

The fourth attribute matched in  $C_1$  is more important than the fifth attribute matched in  $C_2$ . This combination matching function and similarity heuristic included in the retrieval model provides the desired requirements of efficient retrieval, term importance, and relevance estimation.

#### *4.2.4 Facet Reordering*

Unlike retrieval schemes for collections of books and documents, the citation order of a software component classification scheme can be rearranged since the physical collocation of components is not required. This ability to be reordered has been used to advantage in the retrieval mechanism design developed in this research where the facet order can be changed by the user to reflect term importance, relevance ranking, and component clustering. An adjustable citation order allows the user to tailor the collection to a specific set of needs, improving system performance. The faceted scheme's simple list structure and generic relationships between categories directly supports the design of this capability.

### **4.3 Design of the Retrieval System**

The retrieval mechanism developed in Section 4.2 forms the nucleus of an automated software component retrieval system. This section presents the other elements of the system's design. The system uses a query language based on the facets of the faceted classification methodology described in Section 3. A user composes queries by listing appropriate attribute values, ordered by decreasing importance. The internal retrieval process embodies three phases reflecting the boolean, vector space, and similarity heuristic

stages of its design. The design decisions and rationale used in developing this automated software retrieval system are given in the following sections.

#### 4.3.1 Control Structure

A modular control structure was selected for the retrieval system to facilitate implementation, testing, and debugging, and to encourage future experimentation. The central control kernel involves two levels of the control hierarchy as shown in Figures 12 and 13. In the first level of the

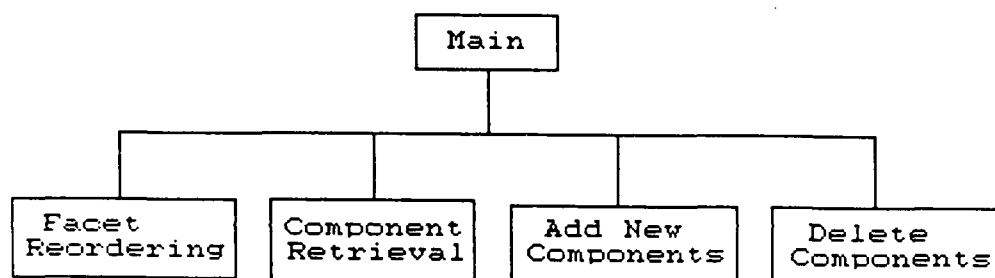


Figure 12. Retrieval System Control Hierarchy (first level)

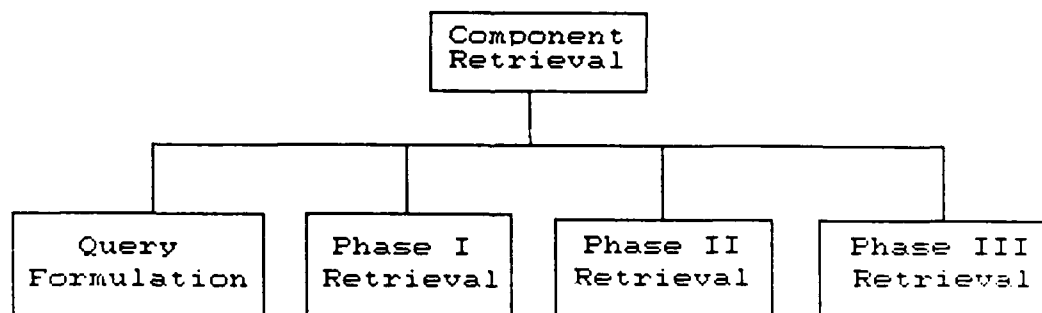


Figure 13. Retrieval System Control Hierarchy (second level)



control hierarchy (Figure 12), the main module provides system initialization and establishes the default facet order (citation order). The user can then specify a different facet order or initiate component retrieval.

The second level of the control structure, illustrated in Figure 13, supports query formulation and the hybrid retrieval mechanism. The retrieval mechanism is designed as a three phase process with each individual phase located in a separate module. This allows any single retrieval phase to be modified or replaced without affecting the rest of the system. Query formulation is also located in a separate module.

The main control logic of the retrieval system is shown in Figure 14 using a pseudo code format. The process begins with a determination of the facet order followed by construction of the query. The retrieval mechanism

```

initialize facet order
if user selects then
  input new facet order
endif
formulate the user query
perform Phase I retrieval {boolean model}
if number retrieved < threshold then
  perform Phase II retrieval {equivalent descriptors}
endif
if number retrieved < threshold then
  perform Phase III retrieval {heuristic model}
endif
if number retrieved > 0 then
  rank by relevance
  output ranked component list
else
  report no components retrieved
endif

```

**Figure 14.** Retrieval System Control Logic

is structured as a three phase process with the execution of each phase controlled by the results of the previous phase. The final step in the process is to rank the retrieved items and output the ordered results. These processes are elaborated in the following sections.

#### *4.3.2 User Interface and Vocabulary Control*

A controlled vocabulary for the index language affects both the user interface and query formulation process. Using the faceted classification schedule as the basis for query construction supports both the design of the user interface and the controlled vocabulary. Each facet or subfacet group can be presented to the user as a set of menus. This aids the user in selecting the most appropriate descriptor for each attribute while, at the same time, limiting the possible choices to only terms allowed in the index language. When a particular facet is unimportant in a query or the user wishes to expand the retrieved set of components, the match-anything characteristic is available as a possible attribute in each set of menus. This design, based on the faceted organization, provides efficient query formulation, vocabulary control, and use of a pre-coordinated index language. After the user forms a query, the system initiates the phased retrieval process.

#### *4.3.3 Phased Retrieval Mechanism*

The retrieval mechanism is integrated into the system design as a three phase process with different phases reflecting the boolean, vector space, and heuristic attributes of its design. After a query is presented to the system, the execution of each phase is internally controlled by the system without further user interaction. The results of each phase controls

the execution of subsequent phases to provide the most relevant components available in the software collection for each query. Phase I is activated first. Designed around the boolean model, Phase I searches the software collection for exact matches of the query tuple. If located, components with matching characteristics are ranked as the most relevant candidates for reuse. A user selected threshold value is used to control the number of retrieved components. If the threshold is not reached during this phase, the second phase of the retrieval process is activated.

Components with attributes that are almost equivalent to the terms in the original query might also satisfy the users' needs. This reverses the role a thesaurus normally plays in retrieval systems (i.e. vocabulary control) in that the thesaurus is used here to expand the query, automatically forming new queries with the almost equivalent descriptors. The new queries are then used to retrieve matching components using a least-important to most-important order. That is, queries with almost equivalent descriptors replacing the least significant facet descriptor are processed first, followed by those replacing the next least significant descriptor, and so on. For example, given the query

$$Q_1 = \langle \text{code, total, real, array, Fortran, PC/DOS, double} \rangle$$

a thesaurus search on the least important descriptor, "double" (from the Precision facet) would locate the almost equivalent attribute "single", and a new query,  $Q_2$ , would be constructed.

$$Q_2 = \langle \text{code, total, real, array, Fortran, PC/DOS, single} \rangle$$

After retrieving components for query  $Q_2$ , almost equivalent terms for the next least significant term, "PC/DOS", would be used. Using the almost equivalent term "MS/DOS", queries  $Q_3$  and  $Q_4$  would be constructed and processed.

$$Q_3 = \langle \text{code, total, real, array, Fortran, MS/DOS, double} \rangle$$

$$Q_4 = \langle \text{code, total, real, array, Fortran, MS/DOS, single} \rangle$$

This process is continued until the component threshold level is reached or until no more equivalent terms can be located in the thesaurus. The thesauri used in many information retrieval systems are created and maintained by the system developers, and cannot be changed by the users. For software component retrieval, a fixed thesaurus might not reflect the experience level or requirements of each user. The capability for each user to modify and specialize the thesaurus would improve retrieval performance.

Phase III contains the similarity heuristic designed in Section 4.2. Beginning with the least important facet descriptor, the retrieval system replaces query terms with the match-anything characteristic. Following each replacement, the query is automatically reprocessed against the component collection. Query  $Q_1$  above would be modified to retrieve components close to the desired attributes by first replacing the least important descriptor, "double", with the match-anything attribute. This generates the new query  $Q_5$ .

$$Q_5 = \langle \text{code, total, real, array, Fortran, PC/DOS, *} \rangle$$

After query  $Q_5$  is processed, the system substitutes the match-anything characteristic for the next least significant descriptor, and so on. During each phase of the retrieval process, duplicate components already found are not re-listed. When the retrieval process is completed, the selected components are presented to the user by order of relevance to the query, namely in the order which they are retrieved.

#### **4.4 Rapid Prototype Implementation**

The development of a faceted software classification methodology and retrieval mechanism to support software reuse was the focus of this research. A rapid prototype was implemented to demonstrate the feasibility of this approach. A data base management system was used for rapid prototype support with the retrieval mechanism added as a front end process. Constructing the prototype involved the selection of appropriate development tools, and implementation of the retrieval control structure, support routines, and user interface. Each of these areas is described below.

##### *4.4.1 Rapid Prototype Support*

The purpose of a rapid prototype is to examine the main features of a system's design as a working system to determine if they satisfy the functional requirements. Prototyping tools are chosen to support a rapid implementation of the system without the usual concerns for efficiency or completeness. The main tool needed to support the rapid prototyping of the software retrieval system should provide data base-like facilities for managing the software collection and should easily manipulate tuple structures to

facilitate implementation of the formal retrieval model. Additionally, a programming capability is needed to implement the phased retrieval mechanism and modular control structure in the retrieval system design.

KnowledgeMan, a relational data base management system developed by Micro Data Base Systems, Inc., was selected as it meets the criteria outlined above [KnowledgeMan 1985]. Developed for IBM PC compatible microcomputers, KnowledgeMan provides the features of a sophisticated data base management system, an internal command language similar to Pascal for programming support, an interactive editor, and a screen interface manager.

#### *4.4.2 Control Structure of the Rapid Prototype*

The main control process for the rapid prototype was implemented directly from the modular control structure design presented in Section 4.3.1. The first level of the control hierarchy provides system initialization, collection management functions, and establishes the default facet order or user specified facet order. The second level of the control structure supports query formulation and the phased retrieval mechanism.

Query formulation uses an index language based on the terms in the faceted classification schedule for data structure algorithms described in Section 3. Using KnowledgeMan's screen interface manager, the user interface was implemented as a set of menus. A user composes a query by selecting the appropriate attribute from each menu. Each menu represents the attributes from one of the classification facets. An example of the facet-based menu approach for query construction is shown in Figure 15. The user

### COMPONENT FORM MENU

Select the **FORM** of the component

1. Requirements
2. Specification
3. Design
4. Test plan
5. Code
6. Commercial package
7. Data

Enter selection number: 5

(a)

Component description: Form = code

Select the main **ACTIVITY** of the component

Simple function (subgroup)

- |                                   |                      |
|-----------------------------------|----------------------|
| 1. add/increment/total/sum        | 14. empty            |
| 2. access                         | 15. encryption       |
| 3. append/attach/concatenate/join | 16. evaluate         |
| 4. close/release/disconnect       | 17. exchange/swap    |
| 5. compare/relate/match/test      | 18. expand           |
| 6. compress                       | 19. format           |
| 7. copy                           | 20. input/read/enter |
| 8. count/enumerate/list           | 21. insert/push      |
| 9. create/produce/generate        | 22. merge            |
| 10. data reduction                | 23. modify/update    |
| 11. decode                        | 24. move             |
| 12. delete/remove/erase           | 25. open/connect     |
| 13. divide/division               | 26. don't care       |

Enter selection number or zero (0) for next screen:   

(b)

**Figure 15.** Facet-based Query Construction: (a) Form menu and (b) Activity menu

first selects a characteristic from the FORM facet illustrated in Figure 15(a). Then the system presents the next menu, in this case the ACTIVITY facet, shown in Figure 15(b), and so on. If a user wants to ignore a particular facet during query construction, a "don't care" option is provided.

The three phase retrieval mechanism is implemented as three separate modules in the rapid prototype with a fourth module to control execution of the other three. Given a query tuple, the retrieval controller first activates the Phase I retrieval process to search the software collection for exact matches of the query. If located, components with matching attributes are ranked as the most relevant. If the user supplied threshold value is not reached during the first phase, the retrieval controller activates the second phase of the retrieval process. The Phase II module searches for items that "almost" match the query. A thesaurus implemented as a KnowledgeMan table is used by the system during this phase to automatically construct and process new queries. The third phase implements the similarity heuristic designed in Section 4.2. This module replaces the least important query descriptor with the match-anything attribute and reprocesses the query. If the threshold has not been reached, the next least important query descriptor is replaced, and so on.

#### *4.4.3 Support Routines*

Although implementation of the rapid prototype followed the main control structure design presented in Section 4.3.1, several additional routines were needed to provide system support functions. A module was developed to interface with the software collection and select the appropriate sets of software clusters for a given query (a screening process). This routine used



the collocation principle of the faceted classification methodology to reduce the search space for the phased retrieval mechanism. Only the selected clusters are passed to the retrieval controller for further processing.

An important design feature of the retrieval system was user reordering of facets to indicate relative importance. A routine was implemented to provide this capability using menu selections for a consistent user interface. The user is asked to indicate the most important category, the next most important, and so on in turn. A global data structure is maintained by this routine to indicate the current facet order.

A support routine was also needed to produce the final ranked output of retrieved software components. This routine is invoked by the retrieval controller when the phased retrieval process is completed. The software components are tagged with relevance indicators during the retrieval process. The output routine uses this information to display the selected components by order of relevance, noting any variances from the original query. A sample of the ranked output is shown in Figure 16 for the query:

q = < code, sort, string, array, Fortran, bubblesort, AOS,  
VAX, 100-999/second, 200-299K bytes, \* >

The component Bubsort, listed first in the output, matches all the given query attributes. The component listed second, Bubsortmt, matches all the query attributes except memory requirement which is measured in words for Bubsortmt instead of bytes as requested in the query. Bubsortmm, the third most relevant component for query q, needs a slightly different amount of memory, and so on.

Components Retrieved

1. Bubsort	
2. Bubsortmt	with memory measurement = words
3. Bubsortmm	with memory amount = 100-199
4. Bubsortpt	with performance time unit = minute
5. Bubsortpr	with performance amount = 1-99
6. Bubsortos	with OS = TOPS
7. Bubsortalg	with algorithm = insertion
8. Bubsort	with language = Basic
9. Bubsortloc	with location = list

**Figure 16.** Ranked Output from the Prototype

Construction and operation of the rapid prototype verified the general principles used in the design of the faceted organization and software retrieval model. The design supports query formation, efficient retrieval, term importance, component clustering, and relevance estimation. A detailed evaluation of the prototype system is presented in Section 6.

## 5. EXPANDING THE SOFTWARE COLLECTION

The faceted classification methodology and component retrieval mechanism produced positive results when applied to the data structure algorithm domain described in Section 3. But other classification schemes devised for small, special collections have worked equally well in many cases [LaMontagne 1961; Foskett 1982]. However, problems and limitations can arise when these schemes are extended or applied to more general situations. The remaining area to be examined in this research then is the expansion of the software collection. Will the faceted methodology be hospitable when adding software components from a different domain?

To provide an answer to this critical question, the faceted classification methodology described in Section 3 was applied to another set of software components. This expansion experiment involved the selection of a different software domain, and application of the faceted methodology and component schedule developed for the data structure domain to representative components from the new domain.

### 5.1 Selecting Another Software Domain

Components from the domain of data structure algorithms were used to construct the original schedule. Since this set of components could have had unknown intrinsic attributes favorable to the faceted technique, the components chosen to test the expansion capabilities of the scheme were selected to differ as much as possible from the data structure collection used for the first test (i.e., not be code fragments and not be oriented around data

structure algorithms). Other requirements desired for the selected set of components were that they be well documented and available for examination to support the classification process. The test collection should also contain a reasonable number of components, since a small collection would not adequately test the modification process or stress the classification scheme. Finally, the selected domain should not have been previously classified using a faceted scheme, since this might predispose the characteristics of the components to the faceted methodology and bias the results of the experiment.

A subset of the International Mathematics Scientific Library [IMSL 1984] was selected to meet these requirements. IMSL components are not code fragments, but complete Fortran subroutines. The IMSL subroutines provide common mathematical and statistical functions, and are not concerned with data structure algorithms. A printed reference manual contains the documentation for each subroutine with the subroutines grouped into chapters using an organization based on enumerative classification. Since the complete IMSL collection contains over 600 components, the test collection was limited to subroutines from Chapter B (Basic Statistics) and Chapter D (Differential Equations). The first step in the faceted methodology is to analyze a representative sample of the software components into their elemental classes. The subroutines selected to form this representative sample are listed below.

Bdcou1  
Bdcou2  
Bdltv  
Bdtab  
Bdtabd

Dcadred  
Dcsqdu  
Dcsqdud  
Dgear  
Dgeard

Dblin  
Dblind  
Dcadre  
Drvte  
Drvted

Bdtrgi  
Bdtrgid  
Bdtrgo  
Dmlind  
Bdtwt  
Drebs

Dgeara  
Dgearad  
Bdtrgod  
Bdtwt  
Becor  
Becord

Dverk  
Dverkd  
Dmlin  
Dpdes  
Dpdesd  
Drebsd

## 5.2 The Modification Process

The addition of new components to a faceted classification scheme requires the same basic analysis process that was used to construct the original schedule. First, the components to be added are analyzed to identify their elemental classes. Then, the existing classification schedule is examined to determine the presence of any of these new elemental classes. For the elemental classes which are already present, no further action is required. If the analysis produces elemental classes that are not in the original schedule, then the schedule must be modified by adding the new terms to the correct facet or subfacet. The following steps outline the modification process.

1. Analyze the software components to be added to the schedule into their elemental classes.
2. Eliminate from the list of new terms any classes which are already present in the original schedule.
3. Group the new elemental classes into facets using the single characteristic principle of division.
4. When necessary, apply different characteristics of division within facets to produce subfacets.
5. Insert the new terms into the original schedule maintaining the citation and foci order.
6. Adjust the combination order for facets (the facet formula) to accommodate any new facets that were added.

This modification process was used in the following section to expand the original classification schedule.

### 5.3 Modifying the Prototype Schedule

The process of adding the IMSL test collection to the original schedule began with an analysis of each software component in the sample to identify its elemental classes. Homographs, synonyms, and hierarchical relationships need to be recognized and handled during this analysis phase, just as they were during construction of the original schedule. Analysis of the new components in the IMSL test collection listed above produced 297 elemental classes. This is within the expected number of terms since the original schedule contains 11 facets. However, a large number of the elemental terms were duplicates (i.e., the Form of every subroutine was "code", the Language of every subroutine was "Fortran"). Removing duplicate terms and eliminating synonyms reduced the list to 38 unique classes. The second step in the modification process removed from the list of new terms any elemental classes which were already present in the original schedule. Ten of the terms were found to already exist in the original schedule, reducing the number of new terms for insertion to 28. The next step involved grouping the new classes into facets. This process produced 4 facets with the new terms grouped as indicated below:

#### Activity (facet)

differential equations  
differentiation  
frequency counting  
integration  
parameter estimation

#### Algorithm (facet)

Adams method  
adaptive romberg  
extrapolation  
finite difference  
Gaussian

partial differential equations  
transgeneration

Focus (facet)

cubic spline  
function  
matrix  
observations  
system of equations

Location (facet)

subprogram

Gears method  
in core  
letter value summary  
method of lines  
multivariate data  
one-way table  
out of core  
quadrature  
runge-kutta  
two-way table

The fourth step, when necessary, divides the facets into subfacets. Within the Focus facet, all the new terms were grouped into the Simple Object subfacet, and the one new term in the Location facet was placed in the Data Structure subfacet.

Activity (facet)

differential equations  
differentiation  
frequency counting  
integration  
parameter estimation  
partial differential equations  
transgeneration

Focus (facet)

*simple object (subfacet)*  
cubic spline  
function  
matrix  
observations  
system of equations

Location (facet)

*data structure (subfacet)*  
subprogram

Algorithm (facet)

Adams method  
adaptive romberg  
extrapolation  
finite difference  
Gaussian  
Gears method  
in core  
letter value summary  
method of lines  
multivariate data  
one-way table  
out of core  
quadrature  
runge-kutta  
two-way table

The final two steps in the modification process integrate the new terms into the original schedule and adjust the combination order when required. The 28 new elemental classes from the IMSL test collection were integrated into the existing schedule for data structure algorithms contained in Appendix 1. The expanded schedule, which grew from 275 to 303 elemental classes, is included as Appendix 2.

#### **5.4 Results and Conclusions**

The hospitality of the faceted classification methodology developed for software components was the focus of this experiment. Overall, the faceted schedule was found to be easy to expand. There was no evidence of any re-classification problems like those normally encountered when expanding enumerative schemes. The modification process was not difficult. Having the original schedule available during the analysis phase simplified the classifier's task in that the original schedule could be used in suggesting facets and elemental classes which characterized the new components. Many of the new terms were already listed in the original schedule, even though the test collection contained components significantly different from those used to form the original schedule. No modifications beyond the addition of new elemental terms were necessary. The original 11 facets were adequate for the expanded collection and no new subfacets were required.

The test collection was a subset of software components from a special library of mathematical routines. Based on the classification process, some interesting observations about the application of a faceted scheme to special software libraries can be made. For example, tailoring the classifica-



tion schedule to fit only the IMSL collection would be an easy process and would produce a compact, effective classification system. Several of the facets in the original schedule could be completely eliminated for this special case. The Form facet, for example, is unnecessary since "code" is the only choice possible for IMSL components. Likewise, Fortran is the only language these routines are written in, making the Language facet superfluous. Finally, anyone seeking a particular IMSL subroutine would already be aware of the general hardware and operating system requirements, so these facets could be eliminated as well. The classification scheme could, therefore, be reduced from 11 to 7 facets. The complete schedule would be extremely concise, requiring only 34 elemental terms. This same tailoring process could be applied to other domain specific software libraries. The faceted classification model can provide a powerful organizational tool for these special software libraries in a small, flexible format.

As additional software components are added to the collection, it appears that increasing numbers of the new elemental classes would already be present in the original schedule. Adding the complete IMSL collection of over 600 subroutines would mostly affect the Algorithm facet since many of the subroutines contain unique mathematical algorithms. This preliminary work with the faceted methodology indicates that many of the other facets in the schedule would quickly stabilize, requiring fewer and fewer additions as more components are added to the collection. The classification process also becomes easier as the classifier's experience grows and more terms become available in the schedule. Adding the IMSL components was not difficult, and it is anticipated that expanding into other domain areas would yield similar results.

## **6. SUMMARY, CONCLUSIONS AND FUTURE WORK**

Studies have shown that reusing existing software can reduce development costs, speed up the development process, and provide a more reliable product. The major goal of this research was to support software component reuse by developing the capability to locate (retrieve) a desired component within a large collection of items. This capability required the development of three cooperating processes: (1) an organization for the software collection, (2) a method for describing components based on that organization, and (3) a mechanism to access (locate and retrieve) the desired component or set of components. A faceted software classification methodology was developed to provide the organization for the collection and to facilitate component description. With the faceted classification scheme providing the underlying structure, techniques from information retrieval theory were used to design a hybrid retrieval mechanism. A rapid prototype was then implemented to determine the feasibility and utility of this approach. This section summarizes the research, presents conclusions which can be drawn from the rapid prototype, and uses the experience and insights gained to make recommendations for future work.

### **6.1 Summary of the Research**

This research can be partitioned into four phases: (1) a detailed study of software reuse, (2) development of a faceted software classification methodology, (3) design of a software retrieval mechanism, and (4) construction of a rapid prototype of the retrieval system. Each of these phases is summarized below.

### *6.1.1 Software Reuse*

During the first phase, a detailed study of software reuse was conducted. Reuse was found to occur in many different forms ranging from the repetitive execution of commercial packages and software development tools, to reuse of code fragments and subroutines, to the reuse of analysis and design knowledge in transformation paradigms. The impetus for each type of reuse, however, was the same: economics. Although reusable software components may be more expensive to develop initially, reusing software can increase productivity and reduce overall development costs as the expense of the reused component is amortized over its uses. Several studies and industrial projects have documented the potential of software reuse. However, these projects have been limited to small-scale applications. The lack of a software classification scheme and a retrieval mechanism are routinely listed as factors limiting current reuse technology. Software reuse will only be practical on a large scale when developers can easily locate a component to reuse. Classification and retrieval capabilities are a cornerstone for successful large-scale software reuse and became the focus of this research.

### *6.1.2 Faceted Methodology*

The second phase of the research involved the design of an organization for software collections. Organization (classification) provides a basis for describing items in the collection and the capability to answer membership questions. Library classification science provided a mature body of knowledge which served as the background for this phase of the research. The history of library classification shows an evolution

progressing from enumerative schemes toward the use of faceted techniques. Enumerative schemes take a subject area and divide it into successively narrower classes listing all the simple and complex subjects which can ever be classified by the scheme. This enumeration (listing) of all possible subjects is one of the major disadvantages of the enumerative method. The more modern classification technique, faceted, is based on a subject analysis and synthesis process. The subjects to be classified are analyzed and divided into their elemental terms. These terms and their relationships are listed in the classification schedule. Synthesis is then used to form more complicated subjects from the elemental terms as needed by assembling them according to the citation order. Since the analysis and synthesis process plays such a central role, faceted schemes are also called analytico-synthetic classification.

The general objectives for classification schemes found in library science were combined with the needs of software classification to form a set of functional requirements for the software scheme. These requirements were used to compare the two classification models. Enumerative schemes were found to be complicated, difficult to automate, time consuming to construct (years or decades), inflexible and difficult to expand without reclassification problems, and unresponsive for rapidly changing subject areas. Faceted classification was designed to overcome many of these disadvantages. A faceted scheme is easy to expand, facilitates automation, supports the description of complex subjects, and has a significantly smaller schedule. The faceted model was used to develop a classification methodology specialized for reusable software components. The methodology employs an analysis technique based on literary warrant to construct the classification schedule. Components to be classified are analyzed into their elemental

terms, which are then grouped into facets and subfacets using the single characteristic of division principle. To complete the schedule, a citation order is established to guide the users during the synthesis process. Synthesis is used to assemble superimposed, complex, and compound classes from the elemental terms in the schedule. This classification methodology provides a flexible descriptive capability for items in the software collection. The faceted methodology was successfully demonstrated on a test collection of components selected from the domain of data structure algorithms and later expanded by the addition of components from the IMSL collection.

### *6.1.3 Retrieval Mechanism*

A retrieval mechanism was developed during the third phase of the research. The field of information retrieval theory provided three models which could be used as the basis for designing the software retrieval mechanism. The probabilistic model was quickly eliminated, leaving the vector space and boolean models as possible candidates. An evaluation of each model indicated certain strengths and weaknesses, but neither model supported all the requirements for software component retrieval. Since some features from each model were needed, a hybrid mechanism was designed using features from both the boolean and vector space models combined with the descriptive structure provided by the faceted methodology. In the design of the formal retrieval mechanism, each component in the software collection was represented as a tuple of attributes, with each attribute being an elemental term from some facet in the faceted classification schedule. Queries were also represented as attribute tuples using a similar format. This structure supported the design

of a relevance measure for ranking retrieved items, a technique for vocabulary control, and the capability to reorder the collection to reflect term importance.

The retrieval mechanism itself was designed as a three phase process. The first phase, designed around the boolean retrieval model, searches the software collection for exact matches of the query tuple. Software components which exactly match the query are considered the best candidates for reuse. The second phase of the retrieval mechanism searches for components with attributes that are almost equivalent to the terms in the original query. A thesaurus containing almost equivalent terms is used to modify the original query, automatically forming new queries with the almost equivalent descriptors. The new queries are then used to retrieve components which might also satisfy the users' needs. Phase III of the retrieval mechanism implements a similarity heuristic designed to locate components "close" to but not exactly matching the query. Beginning with the least important facet descriptor, the similarity heuristic successively replaces query terms with a "match-anything" characteristic. Following each replacement, the modified query is reprocessed against the component collection. Thus, the similarity heuristic selects software components which match the important attributes in the query, although they may not match every attribute.

#### *6.1.4 Rapid Prototype*

A rapid prototype of the retrieval system was implemented based upon this design using KnowledgeMan, a relational data base management system, as the implementation vehicle. Query formulation in the prototype

used an index language based on the terms in the faceted classification schedule. Using KnowledgeMan's screen interface manager, the user interface was implemented as a set of menus with a user composing a query by selecting the appropriate attribute from each facet's menu. The query is then evaluated using the phased retrieval mechanism described above which returns a list of reusable components to the user ordered by relevance. A user selected threshold value controls the number of components selected. The rapid prototype provided an arena in which to examine the retrieval mechanism.

## **6.2 Results and Conclusions**

Conclusions can be drawn from several different areas of this research. The rapid prototype demonstrated the feasibility of the faceted classification methodology and hybrid software retrieval model. The faceted scheme provided a supportive and flexible organization for the software collection and the retrieval model produced encouraging results using the test collection. Specific results and conclusions are presented in the following sections.

### *6.2.1 Rapid Prototype*

The purpose of the rapid prototype was to examine the utility of a faceted classification methodology and hybrid retrieval mechanism as the basis for software component retrieval. Implementation and operation of the prototype confirmed the general principles used in selecting the faceted scheme and in designing the retrieval mechanism. Description synthesis seems to support the formation of specific queries customized to an

appropriate level of precision for each component collection. Organized as separate lists of elemental terms, faceted classification does facilitate automation. Translating the scheme into a working prototype was a straightforward process. Tying the retrieval process to facet order directly supported term importance and created an instrument for relevance measurement.

Using a data base system to construct the rapid prototype allowed the system to be implemented in less than two weeks and supported the primary functions of the retrieval mechanism. However, execution was somewhat slow as expected with the overhead of frequent service calls to the data base system and screen manager. Using the KnowledgeMan screen manager to implement the user interface turned out to be a poor implementation decision. The construction of each screen (menu) was a tedious process and the static nature of each resulting menu did not support the flexibility of the faceted scheme.

Most existing collections of reusable software components rely on printed reference manuals for locating a desired component, and therefore, could not be compared to this automated retrieval system. Information (document) retrieval systems are usually evaluated and compared in terms of recall and precision with measurements in the area of 30% recall and 70% precision considered as good. The small test collection of software components developed during this research precludes a valid statistical study of recall and precision, but preliminary results appear to indicate improvement over traditional document retrieval systems. The precise index language and controlled vocabulary provided by the faceted scheme



may contribute to the retrieval capability. Forming a hybrid retrieval mechanism using the most effective techniques from the boolean and vector space retrieval models added to the prototype's capabilities by combining the efficiency of direct attribute matching with term importance indicators and relevance ranking.

As a preliminary test of the prototype's performance, the following evaluation was performed on approximately 30 sets of queries. First, a query was manually evaluated against the component collection to determine the most relevant software components and their associated order of relevance. Then, an identical query was processed by the prototype system. In each test, the prototype produced results matching those from the manual evaluation. While the small test collection cannot provide irrefutable results, these tests support the design decisions made and indicate a potential for similar results using larger software collections.

Query formulation based on the synthesis process of faceted classification provided a simple but effective user interface. Selecting the appropriate descriptors from lists of elemental terms in each facet eliminated the need for complex query construction rules or user expertise concerning retrieval languages. Adjusting the citation order to fit individual user needs appears to provide added user support by tailoring the organization of the software collection to better fit a particular query.

### *6.2.2 Faceted Classification*

The faceted classification methodology is a useful organization technique for collections of reusable software components, since the

synthesis capability it provides is well suited to the description of detailed and complex subjects. A faceted classification system is easy to modify and expand as was demonstrated in Section 5 when the IMSL subroutines were added to the test collection. Faceted schemes also facilitate automation. The simple relationships between facets, a schedule structure composed of separate lists of terms, and a well defined combination order for synthesis of complex subjects can be directly translated into a computer model.

Faceted schemes are quite flexible and can be adapted to different collections or different users. The adjustable citation order implemented in the rapid prototype provided dramatic results, indicating how effective this technique can be in improving retrieval performance. Tailoring the component collection to fit the user's requirements supports effective query formulation and efficient retrieval. The following example illustrates how retrieval can be affected by modifying the citation order.

The default facet order, shown below, is provided by the prototype system and corresponds to the classification schedule developed in Section 3.

Form  
Activity  
Focus  
Location  
Language  
Algorithm  
Operating system  
Hardware  
Performance rate  
Memory requirement  
Precision

Using this default order, a query with the following characteristics was constructed and processed:

$G_1 = \langle \text{code, sort, string, array, Fortran, bubblesort, AOS,}$   
 $\text{Vax, 100-999/second, 200-299K bytes, * } \rangle$

The ranked output obtained from query  $G_1$ , shown in Figure 17(a), lists 9 software components by decreasing order of relevance. Then the facet order was modified to indicate a different term importance as follows

Form  
 Activity  
 Focus  
 Precision  
 Memory requirement  
 Performance rate  
 Hardware  
 Operating system  
 Algorithm  
 Language  
 Location

and the same query was reprocessed. The new component ranking shown in Figure 17(b) has changed significantly. The component Bubblesortloc which was ranked as only the ninth most relevant component for the first query, became the second most relevant component for the same query using the modified facet order. This demonstrates the potential power and flexibility an adjustable citation order can provide for software component retrieval. A user can easily specify different query descriptor importance by simply reordering the characteristics without the need for complex attribute weighting schemes.

### Components Retrieved

- |                  |                                     |
|------------------|-------------------------------------|
| 1. Bubblesort    |                                     |
| 2. Bubblesortmt  | with memory measurement = words     |
| 3. Bubblesortmm  | with memory amount = 100-199        |
| 4. Bubblesortpt  | with performance time unit = minute |
| 5. Bubblesortpr  | with performance amount = 1-99      |
| 6. Bubblesortos  | with OS = TOPS                      |
| 7. Bubblesortalg | with algorithm = insertion          |
| 8. Bubblesortb   | with language = Basic               |
| 9. Bubblesortloc | with location = list                |

(a)

### Components Retrieved

- |                  |                                     |
|------------------|-------------------------------------|
| 1. Bubblesort    |                                     |
| 2. Bubblesortloc | with location = list                |
| 3. Bubblesortb   | with language = Basic               |
| 4. Bubblesortalg | with algorithm = insertion          |
| 5. Bubblesortos  | with OS = TOPS                      |
| 6. Bubblesortpr  | with performance amount = 1-99      |
| 7. Bubblesortpt  | with performance time unit = minute |
| 8. Bubblesortmm  | with memory amount = 100-199        |
| 9. Bubblesortmt  | with memory measurement = words     |

(b)

**Figure 17.** Ranked Retrieval Output: (a) from default citation order and (b) from modified citation order.

### 6.2.3 *Retrieval Mechanism*

A formal retrieval model and phased retrieval system were designed using a combination of techniques from the boolean and vector space information retrieval models. This hybrid design resulted in a retrieval mechanism providing efficient retrieval via direct attribute matching as in the boolean retrieval model, where the terms in the query tuple are matched with the attributes in each software component tuple. Efficient retrieval was also supported by item clustering which is often used in the vector space retrieval model to reduce the search space. Component clustering was provided in this design by the facet groupings in the faceted classification schedule. The citation order (or user supplied facet order) from the faceted schedule was used to indicate term importance (weighting) in the query tuples, and was combined with the similarity heuristic to provide relevance estimation. The feasibility of this design was demonstrated by the construction and operation of a rapid prototype system with the retrieval tests described in Section 6.2.1 indicating the utility of the retrieval mechanism.

### 6.3 **Future Work**

This research has shown that an effective organizational scheme and retrieval system can be developed for reusable software collections. It provides a beginning and suggests several areas for enhancements and further research.

### *6.3.1 System Enhancements*

The rapid prototype verified the general principles used in developing the faceted classification methodology and in designing the retrieval mechanism. However, several extensions can be suggested in both areas to overcome current limitations and enhance future work with the system. The retrieval system, constructed as a rapid prototype, is understandably inefficient. Building the rapid prototype as a front end to a data base management system created severe overhead problems. Every system action required several calls to the data base system and then internal manipulation of the corresponding results. To support further experimentation with larger software collections, the design should be re-implemented as a complete system using standard software engineering techniques.

Retrieval systems are usually evaluated and compared in terms of recall and precision. Although the preliminary results were encouraging, the small test collection developed during this research prevented a valid study of these measures. This suggests several possible extensions. First, the faceted classification methodology should be applied to more software components, especially those from other domains. This would provide a suitable number of components for testing the retrieval system and extend the range of knowledge concerning the faceted methodology and its utility. With a larger software collection available, a second experiment could examine the recall and precision performance of the retrieval mechanism.

In Section 5, the expandability of the faceted scheme was examined by adding subroutines from the IMSL collection to the original test schedule.

This experiment provided an interesting observation about the application of a faceted scheme to special software libraries. The general set of software facets can be tailored to fit a special library for a given domain by eliminating the unnecessary facets. For the IMSL components, it was noted that the Form, Language, Hardware, and Operating System facets could be eliminated. The retrieval system could be extended to automate this "subsetting" capability for building reusable libraries for specific domains. This would provide better user support and reduce the system's search space.

### *6.3.2 Additional Research*

The results of this research have been encouraging, but there is a need for additional research to make the faceted methodology and retrieval model applicable to large-scale software reuse. Several areas for future work including a classification assistant, organization for other software development products, and software retrieval reuse metrics are suggested in the following sections.

#### *6.3.2.1 A Classification Assistant*

The manual construction of a faceted classification schedule for a large software collection would be a labor-intensive process. Automating the construction or modification of a faceted schedule could make the scheme more practical. For example, the test collection developed during this research contained 11 facets. Analyzing 30 new software components to be added to this schedule would produce over 300 elemental terms. An automated assistant (possibly knowledge-based) could aid the classifier by identifying homographs, synonyms, and duplicate terms (similar capabilities

exist in many editors and word processors). The assistant might also suggest various facet and subfacet groupings based on different user specified characteristics of division. This capability could even be extended to automate production of the interface menus used for query formulation.

#### *6.3.2.2 Other Software Development Products*

The faceted schedule developed during this research was oriented toward source code components. Additional work is needed to test the classification methodology on other products from the software development process such as specifications, requirements, and designs. Reuse of these more abstract forms could produce potentially larger benefits. One goal of this additional research might be to determine if one classification schedule would be adequate for all such products or would separate organizations be required for each different development product.

#### *6.3.2.3 Reuse Environment Integration*

The effective application of software reuse on a large scale will require more than just a retrieval system for locating components. A support environment is needed to integrate all the tools necessary for effective software reuse. The retrieval model and classification methodology designed during this research should be integrated into an existing reusable software development environment and evaluated for its utility and effectiveness. This evaluation could include not only measurements such as precision and recall, but also studies comparing the time spent searching for and modifying a component for reuse with the time it would take to develop a new component instead. The ReadyCode system



at Raytheon [Lanergan and Grasso 1983, 1984] and the Reusable Code system at Hartford Insurance [Cavaliere and Archambeault 1983] are possible candidates for this type of research.

#### *6.3.2.4 Thesaurus Construction*

Section 4 described the use of "almost equivalent" descriptors during the second phase of the retrieval process to automatically generate new queries. These descriptors were selected from a thesaurus which contained almost equivalent terms for some of the original query attributes. For the rapid prototype implemented during this research, the thesaurus of almost equivalent terms was constructed by hand. Two areas for additional research are suggested concerning this thesaurus. First, the construction of the thesaurus could be automated to improve the construction process and produce a richer, more complete thesaurus. The second area concerns specialization of the thesaurus for individual users. Depending on the software domain to be reused, the application area, or user experience level, a user may want to use different almost equivalent terms for some descriptors instead of those provided in the original thesaurus. A knowledge-based approach might be useful for automating the specialization process if the general guidelines (rules) could be identified.

#### *6.3.2.5 Retrieval Based Upon Reuse Metrics*

To support large-scale software reuse, the reusable parts collection may contain thousands of software components. Reuse metrics might be useful in this context to help guide the retrieval mechanism to the most relevant components for a particular query and user. Retrieval based upon

reuse metrics might become a fourth phase in the retrieval mechanism designed during this research. Using the components selected as reuse candidates by the first three retrieval phases, the fourth phase could further refine the selection process by using the reuse metrics as evaluation measures. A limited study of reuse metrics as retrieval indicators was reported by Prieto-Diaz [1985] suggesting such metrics as lines of code, the number of variable references, and the number of subroutine calls. For example, given two software components selected for reuse by the first three phases of the retrieval process with identical relevance measures, the reuse metrics might select one as a better candidate by assuming it is less complex because it has a smaller number of variable references. This basic beginning provides the impetus for additional work in reuse metrics for retrieval which might provide a valuable tool for software retrieval technology.

Based on these initial experiences, the faceted classification methodology and hybrid retrieval mechanism appear to provide an effective retrieval system for reusable software components. This work serves as a first step and encourages additional research to support software reuse.

## REFERENCES

- AFIP 1980. *Taxonomy of Computer Science and Engineering*. Compiled by the AFIPS Taxonomy Committee, AFIPS Press.
- Afshar, S. K. 1985. Software Reuse Via Source Transformations. In *Proceedings COMPSAC 85 Computer Software & Applications Conference*, IEEE Computer Society Press, New York, 54-61.
- Balzer, R., Cheatham, T. E. Jr. and Green, C. 1983. Software Technology in the 1990's: Using a New Paradigm. *IEEE Computer*, Nov, 39-45.
- Balzer, R. 1985. A 15 Year Perspective on Automative Programming. *IEEE Transactions on Software Engineering*, SE-11, No. 11(Nov), 1257-1268.
- Barstow, D. R. 1979. An Experiment in Knowledge-Based Automatic Programming. *Artificial Intelligence*, Vol. 12, No.2(Aug), 73-119.
- Bartschi, M. and Frei, H. P. 1983. Adapting a Data Organization to the Structure of Stored Information. In *Information Retrieval Research* edited by R. N. Oddy, S. E. Robertson, C. J. Van Rijsbergen and P. W. Williams. Butterworth & Co., Ltd, London, 62-79.
- Bartschi, M. 1985. An Overview of Information Retrieval Subjects. *IEEE Computer*, Vol. 18, No. 5(May), 67-84.
- Batz, J. C., Cohen, P. M., Redwine, S. T. and Rice, J. R. 1983. The Application-Specific Task Area. *IEEE Computer*, Vol. 16, No. 11(Nov), 78-85.
- Baumel, L. S., Machovec, K. W., and Sayrs, B. G. 1986. Application Language Issues Report. Internal Lockheed Missiles & Space Company Report D-060736, (Jan), Austin, TX.
- Biggerstaff, T. J. 1984. Foreward, *IEEE Transactions on Software Engineering*, SE-10, No. 5(Sep), 474-476.
- Biswas, G., Subramanian, V. and Bezdek, J. C. 1985. A Knowledge Based System Approach to Document Retrieval. In *The Second Conference of Artificial Intelligence Applications* (Miami Beach, Fl, Dec 11-13), IEEE Computer Society Press, New York, 455-460.
- Blair, D. C. and Maron, M. E. 1985. An Evaluation of Retrieval Effectiveness for a Full-Text Document Retrieval System. *Communications of the ACM*, Vol. 28, No. 3(Mar), 289-299.
- Bliss, H. E. 1985. A Bibliographic Classification: Principles and Definitions. In *The Theory of Subject Analysis*, edited by L. M. Chan, P. A. Richmond and E. Svenonius, Libraries Unlimited, Inc., Littleton, CO., 75-85.

- Bloomberg, M. and Weber, H. 1976. *An Introduction to Classification and Number Building in Dewey*. Libraries Unlimited, Inc., Littleton, CO.
- Boehm, B. W. 1981. *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ.
- Boehm, B. W. and Standish, T. A. 1983. Software Technology in the 1990's: Using an Evolutionary Paradigm. *IEEE Computer*, Vol. 16, No. 11(Nov), 30-37.
- Boehm, B. W., Penedo, M. H., Stuckle, E. D., and Willams, R. D. 1984. A Software Development Environment for Improving Productivity. *IEEE Computer*, Vol. 17, No. 6,(June), 30-42.
- Boisvert, R. F., Howe, S. E. and Kahaner, D. K. 1983. The GAMS Classification Scheme for Mathematical and Statistical Software. *SIGNUM Newsletter*, Vol. 18, No. 1(Jan), 10-18.
- Boisvert, R. F., Howe, S. E. and Kahaner, D. K. 1985. GAMS: A Framework for the Management of Scientific Software. *ACM Transactions on Mathematical Software*, Vol. 11, No. 4(Dec), 313-355.
- Bollmann, P., Konrad, E. and Zuse, H. 1983. FAKYR - A Method Base System for Education and Research in Information Retrieval. In *Research and Development in Information Retrieval*, edited by Gerard Salton and Hans-Jochen Schneider, Springer-Verlag, Berlin, 11-19.
- Bolstad, J. 1975. A Proposed Classification Scheme for Computer Program Libraries. *SIGNUM Newsletter*, Vol. 10, No. 3(Nov), 32-39.
- Booch, G. 1986. *Software Engineering with Ada*, Benjamin/Cummings Publishing Company, Menlo Park, CA.
- Bookstein, A. 1980. Fuzzy Requests: An Approach to Weighted Boolean Searches. *Journal of the American Society for Information Science*, Vol. 31, No. 4(July), 240-247.
- Bookstein, A. 1981. A Comparison of Two Weighting Schemes for Boolean Retrieval. In *Information Retrieval Research*, edited by R. N. Oddy, S. E. Robertson, C. J. Van Rijsbergen and P. W. Williams. Butterworth & Co., Ltd., London, 23-34.
- Bookstein, A. 1983. Explanation and Generalization of Vector Models in Information Retrieval. In *Research and Development in Information Retrieval*, edited by Gerard Salton and Hans-Jochen Schneider, Springer-Verlag, Berlin, 118-132.
- Boyle, J.M. 1980. Software Adaptability and Program Transformation. In *Software Engineering* edited by H. Freeman and P. M. Lewis, Academic Press, New York, 75-93.

- Boyle, J.M. and Muralidharan, M. N. 1984. Program Reusability Through Program Transformation. *IEEE Transactions on Software Engineering*, SE-10, No. 5(Sep), 574-588.
- Buchanan, B. 1979. *Theory of Library Classification*. K.G. Saur Publishing, Inc., New York.
- Buell, D.A. and Kraft, D.H. 1981a. Threshold Values and Boolean Retrieval Systems. *Information Processing and Management*, Vol. 17, No. 3, 127-136.
- Buell, D.A. and Kraft, D.H. 1981b. A Model for a Weighted Retrieval System. *Journal of the American Society for Information Science*, Vol. 32, No.3(May), 211-216.
- Buell, D. A. and Kraft, D. H. 1983. LIARS: A Software Environment for Testing Query Processing Strategies. In *Research and Development in Information Retrieval*, edited by Gerard Salton and Hans-Jochen Schneider, Springer-Verlag, Berlin, 20-27.
- Burton, B. A., Aragon, R. W., Baily, S. A., Koehler, K. D. and Mayes, L. A. 1987. The Reusable Software Library. *IEEE Software*, Vol. 4, No. 4(July), 25-33.
- Cavaliere, M. J. and Archambeault, P. J. Jr. 1983. Reusable Code at the Hartford Insurance Group. In *Proceedings of the Workshop on Reusability in Programming* (Newport, RI, Sep 7-9), 273-278.
- Chan, L. M. 1981. *Cataloging and Classification*. McGraw-Hill Book Company, New York.
- Chandersekaran, C. S. and Perriens, M. P. 1983. Towards an Assessment of Software Reusability. In *Proceedings of the Workshop on Reusability in Programming* (Newport, RI, Sep 7-9), 179-182.
- Comaromi, J. P. 1976. The Historical Development of the Dewey Decimal Classification System. In *Major Classification Systems: The Dewey Centennial*, edited by K. L. Henderson, University of Illinois Graduate School of Library Science, Ill, 17-31.
- Croft, W. B. and Ruggles, L. 1983. The Implementation of a Document Retrieval System. In *Research and Development in Information Retrieval*, edited by Gerard Salton and Hans-Jochen Schneider, Springer-Verlag, Berlin, 28-37.
- Curtis, B. 1983. Cognitive Issues in Reusability. In *Proceedings of the Workshop on Reusability in Programming* (Newport, RI, Sep 7-9), 192-197.
- Deutsch, P. L. 1983. Reusability in the Smalltalk-80 Programming System. In *Proceedings of the Workshop on Reusability in Programming* (Newport, RI, Sep 7-9), 192-197.

- Doszkocs, T. E. 1983. From Research to Application: The CITE Natural Language Information Retrieval System. In *Research and Development in Information Retrieval*, edited by Gerard Salton and Hans-Jochen Schneider, Springer-Verlag, Berlin, 251-262.
- Druffel, L., Redwine, S. and Riddle, W. 1983. The STARS Program: Overview and Rationale. *IEEE Computer*, Vol. 16, No. 11(Nov), 21-29.
- Dumlao, M. and Cook, S. 1983. Cataloging Software. *Special Libraries*, Vol. 74, No. 3(July), 240-245.
- Embley, D. W. and Woodfield, S. N. 1987. A Knowledge Structure for Reusing Abstract Data Types. In *Proceedings of the Ninth International Conference on Software Engineering*, IEEE Computer Society Press, New York, 360-368.
- Faloutsos, C. 1985. Access Methods for Text. *Computing Surveys*, Vol. 17, No.1(Mar), 49-74.
- Fickas, S. F. 1985. Automating the Transformational Development of Software. *IEEE Transactions on Software Engineering*, SE-11, No. 11 (Nov), 1268-1277.
- Foskett, A. C. 1982. *The Subject Approach to Information*. Linnet Books, Hamden, CT.
- Freeman, P. 1983. Reusable Software Engineering: Concepts and Research Directions. In *Proceedings of the Workshop on Reusability in Programming* (Newport, RI, Sep 7-9), 2-16.
- Gargaro, A. and Pappas, T. L. 1987. Reusability Issues and Ada. *IEEE Software*, Vol. 4, No. 4(July), 43-51.
- Gladney, H. M. 1983. A CONCISE Experiment in Program Reusability. In *Proceedings of the Workshop on Reusability in Programming* (Newport, RI, Sep 7-9), 207-214.
- Goodell, M. 1983. Quantitative Study of Functional Commonality in a Sample of Commercial Business Applications. In *Proceedings of the Workshop on Reusability in Programming* (Newport, RI, Sep 7-9), 279-286.
- Gopinath, M. A. 1972. The Colon Classification. In *Classification in the 1970's*, edited by Arthur Maltby, Linnet Books, Hamden, CT, 53-86.
- Grabow, P. C., Noble, W. B. and Huang, C. 1984. *Reusable Software Implementation Technology Reviews*, Hughes Aircraft Company, Fullerton, California, N66001-83-D-0095.
- Green, C. 1976. The Design of the PSI Program Synthesis System. In *Proceedings of the Second International Conference on Software Engineering* (San Francisco, CA, October 13-15), 4-18.

- Green, C., Gabriel, R. P., Kant, E., Kedzierski, B. J., McCune, B. R., Phillips, J. V., Tappel, S. T., and Westfold, S. J. 1979. Results in Knowledge Based Program Synthesis. In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence* (Tokyo, Japan Aug 20-23) IJCAI, 342-344.
- Green, C., Luckham, D., Balzer, R., Cheatham, T. and Rich, C. 1983. Report on a Knowledge-Based Software Assistant. *Report KESU.83.2*, Kestrel Institute, Palo Alto, CA.
- Hafner, C. D. 1981. Representation of Knowledge in a Legal Information Retrieval System. In *Information Retrieval Research* edited by R. N. Oddy, S. E. Robertson, C. J. Van Rijsbergen and P. W. Williams. Butterworth & Co., Ltd., London, 139-153.
- Harrison, W. 1986. A Program Development Environment for Programming By Refinement and Reuse. In *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, Vol. 2A, Western Periodicals Company, North Hollywood, CA, 459-469.
- Horowitz, E. and Munson, J. B. 1984. An Expansive View of Reusable Software. *IEEE Transactions on Software Engineering*, SE-10, No. 5(Sep), 477-487.
- Huang, C. 1985. Reusable Software Implementation Technology: A Review of Current Practice. In *Proceedings COMPSAC 85 Computer Software & Applications Conference*, IEEE Computer Society Press, New York, 207.
- Ichbiah, J. D. 1983. On the Design of Ada. In *Proceedings of the IFIP 9th World Computer Congress* (Paris, France, Sep 19-23), Elsevier Science Publishing Co., Inc., New York, 1-10.
- IDS 1980. *International Directory of Software 1980-81*. Computing Publications Ltd., England.
- IDS 1982. *International Directory of Software 1982-83*. Computing Publications Ltd., England.
- IMSL Library Reference Manual, 1984. IMSL, Inc., Houston, TX.
- Jones, T. C. 1984. Reusability in Programming: A Survey of the State of the Art. *IEEE Transactions on Software Engineering*, SE-10, No. 5(Sep), 488-493.
- Jones, B., Litvintchouk, S., Mungle, J., Krasner, H., Mellby, J. and Willman, H. 1985. Issues in Software Reusability. *ACM SIGSOFT Software Engineering Notes*, Vol. 10, No. 2(Apr), 108-109.
- Kaiser, G. E. and Garlan, D. 1987. Melding Software Systems from Reusable Building Blocks. *IEEE Software*, Vol. 4, No. 4(July), 17-24.

- Kant, E. and Barstow, D. R. 1981. The Refinement Paradigm: The Interaction of Coding and Efficiency Knowledge in Program Synthesis. *IEEE Transactions on Software Engineering*, SE-7, No. 5(Sep), 458-471.
- Kernighan, B. W. 1984. The UNIX System and Software Reusability. *IEEE Transactions on Software Engineering*, SE-10, No. 5(Sep), 513-518
- KnowledgeMan Version 2.0 Reference Manual*, 1985. Micro Data Base Systems, Inc., Lafayette, IN.
- Knuth, D. E. 1973. *The Art of Computer Programming, Volume 1 Fundamental Algorithms*. Addison-Wesley Publishing Company, Reading, MA.
- Kraft, D. H. and Buell, D. A. 1983. Fuzzy Sets and Generalized Boolean Retrieval Systems. *International Journal of Man-Machine Studies*, Vol. 19, No. 1(July), 45-56.
- LaMontagne, L. E. 1961. *American Library Classification*. The Shoe String Press, Inc., Hamden, CT.
- Lanergan, R. G. and Grasso, C. A. 1983. Reusable Designs and Code: A Strategy for Designing Software with Maintenance in Mind. In *Software Maintenance Workshop Record* (Naval Postgraduate School, Dec 6-8), IEEE Computer Society Press, New York, 55-56.
- Lanergan, R. G. and Grasso, C. A. 1984. Software Engineering with Reusable Designs and Code. *IEEE Transactions on Software Engineering*, SE-10, No. 5(Sep), 498-501.
- Langridge, D. 1973. *Approach to Classification*. Linnet Books, Hamden, CT.
- Ledbetter, L. and Cox, B. 1985. Software-ICs A Plan for Building Reusable Software Components. *Byte*, June, 307-316.
- Ledgard, H. 1981. *Ada An Introduction*. Springer-Verlag, New York.
- Lenz, M., Schmid, H. A. and Wolf, P. F. 1987. Software Reuse through Building Blocks. *IEEE Software*, Vol. 4, No. 4(July), 34-42.
- Litvintchouk, S. D. and Matsumoto, A. S. 1984. Design of Ada Systems Yielding Reusable Components: An Approach Using Structured Algebraic Specifications. *IEEE Transactions on Software Engineering*, SE-10, No. 5(Sep), 544-551.
- Lubars, M. D. 1986. Code Reusability in the Large Versus Code Reusability in the Small. *ACM SIGSOFT Software Engineering Notes*, Vol. 11, No.1(Jan), 21-28.



- Maron, M. E. 1983. Probabilistic Approaches to the Document Retrieval Problem. In *Research and Development in Information Retrieval*, edited by Gerard Salton and Hans-Jochen Schneider, Springer-Verlag, Berlin, 98-107.
- Mathis, R. F. 1986. The Last 10 Percent. *IEEE Transactions on Software Engineering*, SE-12, No. 6(June), 705-712.
- Matsumoto, Y., Sasaki, O., Nakajima, S., Takezawa, K., Yamamoto, S., and Tanaka, T. 1981. SWB System: A Software Factory. In *Software Engineering Environments* edited by H. Hunke. North-Holland Publishing Company, New York, 305-318.
- Matsumoto, Y. 1983. Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels. In *Proceedings of the Workshop on Reusability in Programming* (Newport, RI, Sep 7-9), 228-234.
- Matsumoto, Y. 1984. Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels. *IEEE Transactions on Software Engineering*, SE-10, No. 5(Sep), 502-512.
- McCune, B. P. 1977. The PSI Program Model Builder: Synthesis of very High-Level Programs. *SIGART-SIGPLAN Symposium on Artificial Intelligence and Programming Languages*, Vol. 12, No. 8(Aug), 130-139.
- McCune, B. P., Tong, R. M., Dean, J. S. and Shapiro, D. G. 1985. RUBRIC: A System for Rule-Based Information Retrieval. *IEEE Transactions on Software Engineering*, SE-11, No. 9(Sep), 939-944.
- McIlroy, M. D. 1976. Mass Produced Software Components. In *Software Engineering Concepts and Techniques*, J. M. Buxton, P. Naur, and B. Randall, Eds, Petrocelli/Charter, Brussels 39, Belgium, 88-98. From the 1968 NATO Conference on Software Engineering.
- Meyer, B. 1982. Principles of Package Design. *Communications of the ACM*, Vol. 25, No. 7(July), 419-428.
- Micheal, M. E. 1976. Summary of a Survey of the use of the Dewey Decimal Classification in the United States and Canada. In *Major Classification Systems: The Dewey Centennial*, edited by K. L. Henderson, University of Illinois Graduate School of Library Science, Urbana-Champaign, 47-58.
- Mills, J. 1972. The Bibliographic Classification. In *Classification in the 1970's*, edited by Arthur Maltby, Linnet Books, Hamden, CT, 25-52.
- Murray, K. J. B. 1986. Knowledge-Based Model Construction: An Automatic Programming Approach to Simulation Modeling. Ph.D. Dissertation, Computer Science Dept., Texas A&M University, College Station, TX.
- Neighbors, J. M. 1984. The Draco Approach to Constructing Software from Reusable Components. *IEEE Transactions on Software Engineering*, SE-10, No. 5(Sep), 564-573.

- Nourani, C. F. and Jones, G. A. 1985. Software Reusability - A Perspective. In *Proceedings of the Eighteenth Annual Hawaii International Conference on System Sciences*, Vol 2, Western Periodicals Company, North Hollywood, CA., 447-456.
- Osborn, J. 1982. *Dewey Decimal Classification, 19th Edition*. Libraries Unlimited, Inc., Littleton, CO.
- Oskarsson, O. 1983. Software Reusability in a System Based on Data and Device Abstractions - A Case Study. In *Proceedings of the Workshop on Reusability in Programming* (Newport, RI, Sep 7-9), 160-166.
- Partsch, H. and Steinbruggen, R. 1983. Program Transformation Systems. *Computing Surveys*, Vol. 15, No. 3(Sep) 199-236.
- Pedersen, J. T. 1984. The Kongsberg Approach To Achieve Reusable Application Software. In *The First International Conference on Computers and Applications* (Beijing, China, June 20-22), IEEE Computer Society Press, New York, 787-794.
- Pollock, G. M. 1985. A Design Methodology and Support Environment for Complexity Metrics Via Reusable Software Parts. Ph.D. Dissertation, Computer Science Dept., Texas A&M University, College Station, TX.
- Pollock, G. M. and Sheppard, S. 1985. The Use of Ada in Implementing a Rapid Prototyping System. In *Proceedings of the Third Annual National Conference on Ada Technology* (Houston, TX, March 20-21), 145-151.
- Prieto-Diaz, R. 1985. A Software Classification Scheme. Ph.D. Dissertation, Department of Information and Computer Science, University of California, Irvine.
- Radecki, T. 1981. A Model of a Document-Clustering-Based Information Retrieval System with a Boolean Search Request Formulation. In *Information Retrieval Research* edited by R. N. Oddy, S. E. Robertson, C. J. Van Rijsbergen and P. W. Williams. Butterworth & Co., Ltd., London, 334-344.
- Radecki, T. 1983. Generalized Boolean Methods of Information Retrieval. *International Journal of Man-Machine Studies*, Vol. 18, No. 5(May), 407-439.
- Ralston, T. 1981. The Proposed New Computing Reviews Classification Scheme. *Communications of the ACM*, Vol. 24, No. 7(July), 419-433.
- Ramamoorthy, C. V., Prakash, A., Tasi, W. and Usuda, Y. 1984. Software Engineering: Problems and Perspectives. *IEEE Computer*, Vol. 17, No.10(Oct), 191-207.
- Ramamoorthy, C. V., Garg, V. and Prakash, A. 1986. Programming in the Large. *IEEE Transactions on Software Engineering*, SE-12, No. 7(July), 769-783.

- Rauch-Hindin, W. B. 1983. Special Series on System Integration. *Electronic Design*, Vol. 31, No. 3(Feb), 176-194.
- Rich, C. and Waters, R. C. 1983. Formalizing Reusable Software Components. *Working paper Number 251*, MIT Artificial Intelligence Laboratory, Cambridge, MA.
- Robertson, S. E., Van Rijsbergen, C. J. and Porter, M. F. 1981. Probabilistic Models of Indexing and Searching. In *Information Retrieval Research* edited by R. N. Oddy, S. E. Robertson, C. J. Van Rijsbergen and P. W. Williams. Butterworth & Co., Ltd, London, 35-56.
- Robertson, S. E. Maron, M. E. and Cooper, W. S. 1983. The Unified Probabilistic Model for IR. In *Research and Development in Information Retrieval*, edited by Gerard Salton and Hans-Jochen Schneider, Springer-Verlag, Berlin, 108-117.
- Salton, G. 1971. *The Smart Retrieval System*. Prentice-Hall, Inc., Englewood Cliffs, NJ.
- Salton, G. 1981. A Blueprint for Automatic Indexing. *ACM SIGIR Forum*, Vol. 16, No. 2(Fall), 8-38.
- Salton, G. and Wu, H. 1981. A Term Weighting Model Based on Utility Theory. In *Information Retrieval Research* edited by R. N. Oddy, S. E. Robertson, C. J. Van Rijsbergen and P. W. Williams. Butterworth & Co., Ltd., London, 9-22.
- Salton, G. 1982. A Blueprint for Automatic Boolean Query Processing. *ACM SIGIR Forum*, Vol. 17, No. 2(Fall), 6-24.
- Salton, G., Fox, E. A. and Wu, H. 1983. Extended Boolean Information Retrieval. *Communications of the ACM*, Vol. 26, No. 12(Dec), 1022-1036.
- Salton, G. and McGill, M. J. 1983. *Introduction to Modern Information Retrieval*. McGraw-Hill Book Company, New York.
- Salton, G. 1986. Another Look at Automatic Text-Retrieval Systems. *Communications of the ACM*, Vol. 29, No. 7(July), 648-656.
- Sammet, J. E. 1982. The New (1982) Computing Reviews Classification System - Final Version. *Communications of the ACM*, Vol. 25, No. 1(Jan), 13-25.
- Sedgewick, R. 1983. *Algorithms*. Addison-Wesley Publishing Company, Reading, MA.
- SHARE Reference Manual*, 1963. IBM Users Group.
- Silverman, B. G. 1985. Software Cost and Productivity Improvements: An Analogical View. *IEEE Computer*, Vol. 18, No. 5(May), 86-96.

- Smith, D. R., Kotik, G. B., and Westfold, S. J. 1985. Research On Knowledge-Based Software Environments at Kestrel Institute. *IEEE Transactions on Software Engineering*, SE-11, No. 11(Nov), 1278-1295.
- Soloway, E. and Ehrlich, K. 1983. What Do Programmers Reuse? Theory and Experiment. In *Proceedings of the Workshop on Reusability in Programming* (Newport, RI, Sep 7-9), 184-191.
- Soloway, E. and Ehrlich, K. 1984. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, SE-10, No.5(Sep), 595-609.
- SPSS 1984. *Statistical Package for the Social Sciences*. McGraw-Hill, New York.
- Standish, T. A. 1983. Software Reuse. In *Proceedings of the Workshop on Reusability in Programming* (Newport, RI, Sep 7-9), 45-49.
- Standish, T. A. 1984. An Essay on Software Reuse. *IEEE Transactions on Software Engineering*, SE-10, No. 5(Sep), 494-497.
- Tajima, D. and Matsubara, T. 1984. Inside the Japanese Software Industry. *IEEE Computer*, Vol. 17, No. 3(Mar), 34-43.
- Taylor, A. G. 1985. *Bohdan S. Wynar Introduction to Cataloging and Classification*. Libraries Unlimited, Inc., Littleton, CO.
- Tenenbaum, A. M. and Augenstein, M. J. 1981. *Data Structures Using Pascal*. Prentice-Hall, Inc., Englewood Cliffs, NJ.
- Topping, P. C. and Baumel, L. S. 1985. System Specification for LMSC's Expert Requirements Expression and System Synthesis Environment (EXPRESS). Internal Lockheed Missiles & Space Company Report (Apr), Austin, TX.
- Van Rijsbergen, C. J. 1979. *Information Retrieval*. Butterworths & Co., Ltd., Boston, MA.
- Vickery, B. C. 1960. *Faceted Classification: A Guide to Construction and Use of Special Schemes*. Aslib, London.
- Wartik, S. P. and Penedo, M. H. 1986. Fillin: A Reusable Tool for Form-Oriented Software. *IEEE Software*, Vol. 3, No. 2(Mar), 61-69.
- Wasserman, A. I. and Freeman, P. 1983. Ada Methodologies: Concepts and Requirements. *ACM SIGSOFT Software Engineering Notes*, Vol. 8, No.1(Jan), 33-50.
- Waters, R. C. 1983. Formalizing Reusable Software Components. *Working Paper Number 251*, MIT Artificial Intelligence Laboratory, Cambridge, MA.

- Waters, R. C. 1985a. KBEmacs: A Step Toward the Programmer's Apprentice. *Technical Report No. 753*, MIT Artificial Intelligence Laboratory, Cambridge, MA.
- Waters, R. C. 1985b. The Programmer's Apprentice: A Session with KBEmacs. *IEEE Transactions on Software Engineering*, SE-11, No. 11(Nov), 1296-1320.
- Wegner, P. 1983. Varieties of Reusability. In *Proceedings of the Workshop on Reusability in Programming* (Newport, RI, Sep 7-9), 30-44.
- Wegner, P. 1984. Capital-intensive Technology and Reusability. *IEEE Software*, Vol. 1, No. 3(July), 7-45.
- Wirth, N. 1976. *Algorithms + Data Structures = Programs*. Prentice-Hall, Inc., Englewood Cliffs, NJ.
- Woodfield, S. N. 1985. Object Oriented Libraries. In *Proceedings of the Phoenix Fourth Annual International Conference on Computers and Communications*, IEEE Computer Society Press, New York, 42-45.
- Yeh, R. T., Roussopoulos, N. and Chu, B. 1984a. Management of Reusable Software. In *IEEE Fall COMPCON 1984*, IEEE Computer Society Press, New York, 311-320.
- Yeh, R. T., Mittermeir, R., Roussopoulos, N. and Reed, J. 1984b. A Programming Environment Framework Based on Reusability. In *Proceedings of the International Conference on Data Engineering* (Los Angeles, CA, April 24-27), IEEE Computer Society Press, New York, 277-280.

## APPENDIX 1

## SOFTWARE COMPONENT SCHEDULE 1

**Form**

requirement  
specification  
design  
test plan  
code  
commercial package  
data

search/find/locate  
sort/order/rank  
start/initialize  
split  
subtract/decrement  
transpose  
traverse

**Activity**

*simple function (subfacet)*  
add/increment/total/sum  
access  
append/attach/concatenate/join  
close/release/match/test  
compare  
copy  
count/enumerate/list  
create/produce/generate  
data reduction  
decode  
delete/remove/erase  
divide/division  
empty  
encryption  
evaluate  
exchange/swap  
expand  
format  
input/read/enter  
insert/push  
merge  
modify/update  
move  
open/connect  
output/print/echo/write/list  
queueing  
remove/pop  
rename

*application (subfacet)*  
accounting  
administration  
banking  
communication  
compiler  
interpreter  
linker  
management  
security  
translator  
transportation  
utilities

**Focus**

*simple object (subfacet)*  
array  
blank/space  
character/alphanumeric  
digit/number  
directory  
expression  
file  
graph  
heap  
integer/number  
key  
line  
list/linked list  
logical

node  
 numeric  
 page  
 pattern  
 pointer  
 queue  
 real/floating point  
 record  
 set  
 stack  
 statement/instruction/sentence  
 string  
 tab  
 table/relation  
 text  
 tree  
 word/name

*area (subfacet)*

accounts payable  
 accounts receivable  
 commercial loan  
 computer languages  
   (see languages)  
 database  
 word processing

**Location**

*data structure (subfacet)*

array  
 buffer  
 directory  
 expression  
 file  
 graph  
 heap  
 line  
 list/linked list  
 node  
 pointer  
 queue  
 record  
 set  
 stack  
 statement/instruction/sentence  
 string  
 table/relation  
 text

tree  
 word/name

*device (subfacet)*

ALU  
 cache  
 channel  
 controller  
 CPU  
 disk  
 joy stick  
 keyboard  
 memory  
 modem  
 mouse  
 network/LAN  
 pipeline  
 plotter  
 printer  
 processor  
 array processor  
 front-end processor  
 parallel processor  
 vector processor  
 optical scanner  
 register  
 screen/CRT/console  
 sensor  
 tape  
 track ball

**Language**

Ada  
 Algol  
 APL  
 Assembler  
 Basic  
 C  
 Cobol  
 Fortran  
 Lisp  
 Pascal  
 PL/1  
 PSL/PSA  
 Prolog  
 Refine

**Algorithm**

absolute  
acyclic  
after  
alaising  
ASCII  
AVL  
balanced  
batch  
before  
best fit  
binary  
bisection  
bubblesort  
content  
deque  
directed  
distributed  
division  
EBCDIC  
enumerate  
EOF  
exchange  
Fibonacci  
FIFO  
first fit  
fixed  
floating  
folding  
hashed  
heapsort  
hierarchical  
index sequential  
indexed  
insertion  
inverted  
inverted list  
least squares  
linked  
midsquare  
multiway  
natural  
network  
next fit  
n-ary  
online  
partition  
parallel  
polyphase  
postorder  
priority

quicksort  
radix  
radix transformation  
random  
recursion  
relational  
relative  
selection  
sequential  
shellsort  
simulation  
straight  
tagged  
undirected  
variable

**Operating system**

AOS  
CICS  
CP/M  
GCOS  
GENERA  
ISIS  
MCP  
MS/DOS  
MVS  
MVT  
RSX11  
TOPS  
TSO  
UNIX  
VS



**Hardware**

700-799  
800-899  
900 or more

Burroughs 76xx  
CDC Cyber  
CDC 66xx  
DEC VAX  
DG Eclipse  
DG Nova  
Honeywell 60xx  
HP 3000  
IBM PC  
IBM System 38  
IBM 30xx  
IBM 43xx  
Intell 8080  
NCR PC  
PDP 11  
Symbolics  
Univac 11xx  
Zilog Z80

*type of memory measurement (subfacet)*  
words  
bytes

**Precision**

single  
double

**Performance rate**

*performance amount (subfacet)*

1-99  
100-999  
1000-1999  
2000-2999  
3000-3999  
4000-4999  
5000 or more

*performance time unit (subfacet)*

per minute  
per second  
per microsecond

**Memory requirements**

*memory amount (subfacet)*

1-99  
100-199  
200-299  
300-399  
400-499  
500-599  
600-699

## APPENDIX 2

## SOFTWARE COMPONENT SCHEDULE 2

**Form**

requirement  
specification  
design  
test plan  
code  
commercial package  
data

open/connect  
output/print/echo/write/list  
parameter estimation  
partial differential equations  
queueing  
remove/pop  
rename  
search/find/locate  
sort/order/rank  
start/initialize  
split  
subtract/decrement  
transgeneration  
transpose  
traverse

**Activity**

*simple function (subfacet)*  
add/increment/total/sum  
access  
append/attach/concatenate/join  
close/release/match/test  
compare  
copy  
count/enumerate/list  
create/produce/generate  
data reduction  
decode  
delete/remove/erase  
differential equations  
differentiation  
divide/division  
empty  
encryption  
evaluate  
exchange/swap  
expand  
format  
frequency counting  
input/read/enter  
integration  
insert/push  
merge  
modify/update  
move

*application (subfacet)*  
accounting  
administration  
banking  
communication  
compiler  
interpreter  
linker  
management  
security  
translator  
transportation  
utilities

**Focus**

*simple object (subfacet)*  
array  
blank/space  
character/alphanumeric  
cubic spline  
digit/number  
directory

expression  
 file  
 function  
 graph  
 heap  
 integer/number  
 key  
 line  
 list/linked list  
 logical  
 matrix  
 node  
 numeric  
 observations  
 page  
 pattern  
 pointer  
 queue  
 real/floating point  
 record  
 set  
 stack  
 statement/instruction/sentence  
 string  
 system of equations  
 tab  
 table/relation  
 text  
 tree  
 word/name

*area (subfacet)*  
 accounts payable  
 accounts receivable  
 commercial loan  
 computer languages  
     (see languages)  
 database  
 word processing

## Location

*data structure (subfacet)*  
 array  
 buffer  
 directory  
 expression  
 file  
 graph

heap  
 line  
 list/linked list  
 node  
 pointer  
 queue  
 record  
 set  
 stack  
 statement/instruction/sentence  
 string  
 subprogram  
 table/relation  
 text  
 tree  
 word/name

*device (subfacet)*  
 ALU  
 cache  
 channel  
 controller  
 CPU  
 disk  
 joy stick  
 keyboard  
 memory  
 modem  
 mouse  
 network/LAN  
 pipeline  
 plotter  
 printer  
 processor  
 array processor  
 front-end processor  
 parallel processor  
 vector processor  
 optical scanner  
 register  
 screen/CRT/console  
 sensor  
 tape  
 track ball

**Language**

Ada  
 Algol  
 APL  
 Assembler  
 Basic  
 C  
 Cobol  
 Fortran  
 Lisp  
 Pascal  
 PL/1  
 PSL/PSA  
 Prolog  
 Refine

**Algorithm**

absolute  
 acyclic  
 Adams method  
 adaptive romberg  
 after  
 alaising  
 ASCII  
 AVL  
 balanced  
 batch  
 before  
 best fit  
 binary  
 bisection  
 bubblesort  
 content  
 deque  
 directed  
 distributed  
 division  
 EBCIDIC  
 enumerate  
 EOF  
 exchange  
 extrapolation  
 Fibonacci  
 FIFO  
*finite difference*  
 first fit  
 fixed  
 floating

folding  
 gaussian  
 Gears method  
 hashed  
 heapsort  
 hierarchical  
 in core  
 index sequential  
 indexed  
 insertion  
 inverted  
 inverted list  
 least squares  
 letter value summary  
 linked  
 method of lines  
 midsquare  
 multivariate data  
 multiway  
 natural  
 network  
 next fit  
 n-ary  
 one-way table  
 online  
 out of core  
 partition  
 parallel  
 polyphase  
 postorder  
 priority  
 quadrature  
 quicksort  
 radix  
 radix transformation  
 random  
 recursion  
 relational  
 relative  
 runge-kutta  
 selection  
 sequential  
 shellsort  
 simulation  
 straight  
 tagged  
 two-way table  
 undirected  
 variable

**Operating system**

AOS  
 CICS  
 CP/M  
 GCOS  
 GENERA  
 ISIS  
 MCP  
 MS/DOS  
 MVS  
 MVT  
 RSX11  
 TOPS  
 TSO  
 UNIX  
 VS

*performance time unit (subfacet)*  
 per minute  
 per second  
 per microsecond

**Memory requirements**

*memory amount (subfacet)*  
 1-99  
 100-199  
 200-299  
 300-399  
 400-499  
 500-599  
 600-699  
 700-799  
 800-899  
 900 or more

**Hardware**

Burroughs 76xx  
 CDC Cyber  
 CDC 66xx  
 DEC VAX  
 DG Eclipse  
 DG Nova  
 Honeywell 60xx  
 HP 3000  
 IBM PC  
 IBM System 38  
 IBM 30xx  
 IBM 43xx  
 Intell 8080  
 NCR PC  
 PDP 11  
 Symbolics  
 Univac 11xx  
 Zilog Z80

*type of memory measurement (subfacet)*  
 words  
 bytes

**Precision**

single  
 double

**Performance rate**

*performance amount (subfacet)*  
 1-99  
 100-999  
 1000-1999  
 2000-2999  
 3000-3999  
 4000-4999  
 5000 or more